ERROR DETECTION, ANALYSIS AND RECOVERY IN XPL BASED COMPILERS

Lyle Vernon Rich

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ERROR DETECTION, ANALYSIS AND RECOVERY
IN
XPL BASED COMPILERS

by

Lyle Vernon Rich

Thesis Advisor: R. C. Bolles

December 1971

Approved for public release; distribution unlimited.

Error Detection, Analysis and Recovery in XPL Based Compilers

by

Lyle Vernon Rich
Ensign, United States Navy
B.S., Illinois State University, 1969

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1971

ABSTRACT

This thesis involves the detection, recovery and/or correction of errors in XPL defined languages. XPL is a compiler generating system based on a (1,1) bounded context parser using (2,1) context to resolve conflicts in the grammar, and an analyzer which produces tables from a BNF description of the grammar for the language. The areas of spelling errors and errors caused by insertion/deletion are covered. Routines for correcting spelling errors in an ALGOL-like language are presented. An expanded syntax analyzer which aids in the production of a data base used by the compiler to correct insertion/deletion errors is also presented. Ideas for implementing this data base in XPL compilers, using heuristics to decrease the size of the insertion sets is also presented.

TABLE OF CONTENTS

ACKNOWLEDGEMENT ---------- 5

I. INTRODUCTION ---------- 6

II. SPELLING CORRECTION ---------- 17

A. EARLY WORK ---------- 17

1. Blair's Work ---------- 17

2. Freeman's Method ---------- 18

3. Morgan's Method ---------- 19

B. CHOOSING AN ALGORITHM ---------- 19

C. DETECTION OF MISSPELLINGS ---------- 20

D. SPELLING CORRECTION ALGORITHM ---------- 21

E. ACTUAL IMPLEMENTATION ---------- 23

F. OTHER CONSIDERATIONS FOR SPELLING CORRECTION ----- 25

III. ERRORS OF INSERTION/DELETION ---------- 27

A. INITIAL COMMENTS ---------- 27

B. PREPARATION OF A DATA BASE ---------- 27

C. INTRODUCTION TO THE ANALYZER ---------- 29

D. THE ANALYZER ---------- 29

E. HEURISTICS ---------- 30

F. RESULTS ---------- 33

IV. CONCLUSION ---------- 37

V. TOPICS FOR FURTHER CONSIDERATION ---------- 38

A. DATA BASE IMPLEMENTATION ---------- 38

B. EXAMINE MORE CONTEXT ---------- 39

C. MULTIPLE INSERTIONS/DELETIONS ---------- 39

D. SPECIAL GRAMMARS ---------- 39

E. HEURISTICS TO ELIMINATE TRIPLES ---------- 39

F. MORE POWERFUL SPELLING CORRECTION ---------- 39

COMPUTER PROGRAMS ---------- 40

LIST OF REFERENCES ---------- 67

INITIAL DISTRIBUTION LIST ---------- 68

FORM DD 1473 ---------- 69

ACKNOWLEDGEMENTS

The research for this thesis was done on an IBM 360/67 and the CP/CMS system at the Naval Postgraduate School Computer Center.

I wish to express my thanks to my thesis advisor, Lt (j.g.) Robert C. Bolles, for his help, encouragement and advice during work on this thesis.

I would also like to thank LT Gary A. Kildall for his help and suggestions.

I. INTRODUCTION

Consider the following two programs, one in an ALGOL-like language and the other in a FORTRAN-like language.

```
BEGIN
     INTEGER SIDE1, SIDE2, ROW1, ROW2;
     READ (SIDE1, SIDEI, ROW1, ROW2);
     SIDE1:=SIDE2*ROWU+ROW2;
     IF (SIDE1 = SIDE2) THEN WRITE (SIDE1);
     FOR I:=SIDE1 UNTL SIDE2 DO
         BEGIM
             WRITE (SIDE1-SIDE2);
             SIDE2=ROWU-SIDE1;
         EMD;
     WRITE (ROW1, ROW2, SIDE1, SIDE2);
END.
```

```
     READ (5, 100) SIDE1, SIDE2, ROW1, ROW2
     SIDE1=SIDE2*ROWU+ROW2
     IF (SIDE1.EQ.SIDE2) WRITE (6, 101) SIDE 1
     DO 10 I=SIDE1, SIDE2
     X=SIDE1-SIDE2
     WRITE (6, 102) X
     SIDE2:=ROWU+SIDE1
10   CONTINU
     WRITE (6, 103) ROW1, ROW2, SIDE1, SIDE2
100  FORMAT ******
101  FORMAT ******
102  FORMAT ******
103  FORMAT ******
     STOP
     EMD
```

Both are simple programs, but would needlessly fail to execute due to errors in keypunching. Both programs have identical errors involving the misspelling of keywords (reserve words) as well as identifiers. They both have ample local context to determine the nature of the misspellings as well as to correct the errors, all of which are of the nature which this expanded XPL system is designed to handle. The ALGOL-like language is rich in information which could be used to correct the misspellings, while the FORTRAN-like language has sufficient information but not in the

easily accessable form of the ALGOL-like language. FORTRAN-like languages are characterized by a minimum of predeclaration, normally only arrays and the preliminary type conventions, while ALGOL-like languages require the predeclaration of all identifiers except labels.

When considering errors one can divide them up into three basic types: 1) simple errors, e.g. misspelled identifiers, single insertions or single deletions, which can be corrected with no effect on the code generation or the execution of the program; 2) errors which can be corrected but may affect the execution of the program, e.g. misspelled identifiers with multiple possible replacements or insertions where multiple insertions were possible but heuristic reduction has produced a single insertion, e.g. the set of arithmetic operators was reduced to plus only. These errors can be corrected and parsing will continue correctly but the execution may be erroneous; and 3) errors which are so severe that the compiler cannot correct them. In these cases the only alternative is the deletion of some text and the termination of code emission or marking the code as unexecutable.

All of the errors in the two initial examples were of type one. These errors lead to needless resubmission and frustration on the part of the student or programmer who submitted them. Among the simple spelling errors in these samples are ROWU (ROW1), BEGIM (BEGIN), SIDEI (SIDE2), EMD (END), and CONTINU (CONTINUE).

Now let us consider what is involved in the more complex problem of correcting errors not related to spelling. The first program contains an example of a simple insertion, the missing colon between SIDE2 and the equal sign in line eight. The second contains an example of a simple deletion, the colon in line seven. With a reasonable amount of effort on

the part of the compiler designer these programs could have run at first submission and the execution could have been error free. Consider the following examples:

```
LEFT = RIGHT * CORNER                    ALGOL
LEFT := RIGHT * CORNER                   FORTRAN
```

These particular cases are also type one errors and are most frustrating to the programmer because they are obvious to anyone with any knowledge of the two languages. The statements simply require the insertion/deletion of a colon between the first identifier and the equal sign.

The following error is an example of class two errors, involving errors where local context is not sufficient to correct the error with absolute certainty.

```
FOR I:=   UNTIL N DO ....                ALGOL
DO 10 I=,N                               FORTRAN
```

These two statements could not be parsed reasonably by any existing compiler. Compilers could, however, use heuristics to decide upon a "reasonable" insertion, make the insertion, and emit an error message. In both cases a default lower limit of one could be established and inserted in the proper positions or allow the default insertion for a number to be inserted and continue.

Consider:

```
INTEGER A B,C,D                          ALGOL
INTEGER*4 A B,C,D                        FORTRAN
```

These two statements require more complex heuristics involving either the insertion of a comma between the A and B or the deletion of the blank and creation of the single identifier AB.

The final class of error is characterized by two parts of the text which do not belong together. Common examples of this error are extra cards in a program or cards out of place, e.g. ELSE A:=B followed by IF A LSS B THEN B:=A.

The basis of this thesis is the XPL compiler generating system, SKELETON. This system is a compiler generating system based on a (1,1) bounded context parser using (2,1) context to resolve conflicts in the grammar, and an analyzer which produces parsing tables from a BNF (Backus Naur Form) description of the grammar for the language.

There are two basic methods for recovering from an error. 1) Use semantics and local context to correct the error as best possible; and 2) Delete text until continuing. It is this second method which is used by the XPL system and which is the least desirable, since it totally eliminates the possibility of a correct parse and the execution. XPL presently recognizes errors in two ways: 1) "ILLEGAL SYMBOL PAIR" caused by the appearance of two terminals together for which no stacking decision exists; 2) "NO PRODUCTION APPLICABLE", a condition occuring when local context requires a reduction but none can be made. This type of system keys on end symbols such as ";" or END and normally deletes input text until one of the key symbols is encountered. They also key on some stack symbols such as <block body> or <statement list> and delete useless material from the stack, although XPL in its present form does this only to a very limited degree. This error recovery technique allows, almost forces, errors to cascade down through good text, thus producing multiple error messages.

The first method, on the other hand, uses local context if possible to correct the error and continue parsing. This can be done by checking

spelling if identifiers or reserve words are present, or by trying to determine if a symbol is missing or if an extra symbol has been inserted.

Consider figure 1, a simple program designed to test this system. The program contains only minor spelling errors. The ALGOL-E compiler [6], on which the BALGOL test compiler was based, using standard XPL error recovery techniques produced 19 errors — 9 illegal symbol pairs, 5 undefined identifiers, 1 invalid for loop index, etc. — and in fact failed to detect all of the errors due to text deletion. The same program using the system presented here detected 23 errors and executed normally (see figure 2).

The problem of making reasonable corrections leads us into the fields of artificial intelligence and pattern recognition. For example, spelling correction is an application of the concepts which have been developed in the area of pattern recognition. Many of these techniques are well suited to the problem of compiler design. The use of heuristics in compilers to correct other errors requires the building of a data base from which the compiler can draw information when it reaches a point where parsing of the program cannot continue. The present parsing tables are such a data base. Optimally this data base must be as small in size as well as organized in such a manner that it is easily accessible. The data base thus becomes the basis for the correction of other than simple misspellings.

The following is a discussion of: 1) a set of spelling correction routines implemented in an XPL compiler, 2) a method for correcting errors caused by insertions and deletions, 3) a new syntax analyzer which provides information useful in handling insertions and deletions, and 4) some suggestions for implementation of this technique in XPL based compilers.

```
    A L G O L - E    C O M P I L A T I O N                          (VERSIO
TODAY IS DECEMBER 3, 1971.  CLOCK TIME = 11:11:51.05.
CARD |BL| SYL |
   1 |  0|   0 |BEGIN LOCAL I,J,K;
   2 |  1|   0 |K:=2;
   3 |  1|   3 |I:=J:=K; WRITE(K,J,I);
   4 |  1|  19 |IF (I LEQ 2) THEN I:=5;
   5 |  1|  30 |K:=J:=I; WRITE(I,J,K);
   6 |  1|  46 | BEGIN LOCAL LYLE,JERRY,MARY,JOHN,JIM,HEINRICH,KARL,KATHY;
   7 |  2|  46 |   BEGIM LOCEL RICH;
                    |
*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <IDENTIFIER>.  LAST PREVIOUS ERROR
PARTIAL PARSE TO THIS POINT IS:  <BLOCK HEAD> <BLOCK BODY> ; <BLOCK HEAD> <IDENT
   8 |  2|  46 |   FOX LILE:=1 STEP 2 UNTIL 2 DO JAM:=1;
RESUME:     |
                  |
*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <IDENTIFIER>.  LAST PREVIOUS ERROR
PARTIAL PARSE TO THIS POINT IS:  <BLOCK HEAD> <BLOCK BODY> ; <IDENTIFIER>
   9 |  2|  46 |   FOR LYLL:=1 STAP 2 UNTIL 2 DO KERL:=2;
RESUME:     |
                  |
*** ERROR, UNDECLARED IDENTIFIER.  LAST PREVIOUS ERROR WAS DETECTED ON LINE 8.
                     |
*** ERROR, ILLEGAL SYMBOL PAIR: <NUMBER> <IDENTIFIER>.  LAST PREVIOUS ERROR WAS
PARTIAL PARSE TO THIS POINT IS:  <BLOCK HEAD> <BLOCK BODY> ; FOR <VARIABLE> : =
  10 |  2|  46 |   FOR JARRY:=1 STEP 2 UNTLI 2 DO KETHY:=3;
RESUME:     |
                  |
*** ERROR, UNDECLARED IDENTIFIER.  LAST PREVIOUS ERROR WAS DETECTED ON LINE 9.
                        |
*** ERROR, INVALID FOR-LOOP INDEX.  LAST PREVIOUS ERROR WAS DETECTED ON LINE 10.
                           |
*** ERROR, ILLEGAL SYMBOL PAIR: <NUMBER> <IDENTIFIER>.  LAST PREVIOUS ERROR WAS
PARTIAL PARSE TO THIS POINT IS:  <BLOCK HEAD> <BLOCK BODY> ; <FOR CLAUSE> <STEP>
  11 |  2|  48 |   MARY:=3;
RESUME:     |
  12 |  2|  51 |   WRITA(MARY);
                    |
*** ERROR, UNDEFINED PROCEDURE.  LAST PREVIOUS ERROR WAS DETECTED ON LINE 10.  *
                        |
*** ERROR, WRONG NUMBER OF PARAMETERS.  LAST PREVIOUS ERROR WAS DETECTED ON LINE
                        |
*** ERROR, ILLEGAL SYMBOL PAIR: <LOWER BOUND> ;.  LAST PREVIOUS ERROR WAS DETECT
PARTIAL PARSE TO THIS POINT IS:  <BLOCK HEAD> <BLOCK BODY> ; <LOWER BOUND>
  13 |  2|  55 |   IF (MERY LST 4) THAN MERY:=9;
RESUME:    |
                     |
*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <IDENTIFIER>.  LAST PREVIOUS ERROR
```

Figure 1. Test Program — ALGOL-E Compiler

```
PARTIAL PARSE TO THIS POINT IS:  <BLOCK HEAD> <BLOCK BODY> ; IF ( <IDENTIFIER>
  14 | 2|  55 |   WRITE(MARY);
RESUME:      |
  15 | 2|  59 |   HEINRICH:=2;
  16 | 2|  62 |   CASS HEINRICK FO
                  |
*** ERROR, ILLEGAL SYMBOL PAIR: <IDENTIFIER> <IDENTIFIER>.  LAST PREVIOUS ERROR
PARTIAL PARSE TO THIS POINT IS:  <BLOCK HEAD> <BLOCK BODY> ; <IDENTIFIER>
  17 | 2|  62 |     BEGIN WRITE(HEINRICK); WRITE(HEINRICK);
RESUME:                                  |                 |
*** ERROR, UNDECLARED IDENTIFIER.  LAST PREVIOUS ERROR WAS DETECTED ON LINE 16.
  18 | 2|  65 |             WRITE(HIENRICH); END;
                                  |
*** ERROR, UNDECLARED IDENTIFIER.  LAST PREVIOUS ERROR WAS DETECTED ON LINE 17.
  19 | 1|  68 |   EMD;
              |
*** ERROR, UNDECLARED IDENTIFIER.  LAST PREVIOUS ERROR WAS DETECTED ON LINE 18.
              |
*** ERROR, ILLEGAL SYMBOL PAIR: <LOWER BOUND> ;.  LAST PREVIOUS ERROR WAS DETECT
PARTIAL PARSE TO THIS POINT IS:  <BLOCK BODY> ; <LOWER BOUND>
  20 | 1|  69 |END
RESUME:    |
  21 | 1|  69 |END
           |
*** ERROR, NO PRODUCTION IS APPLICABLE.  LAST PREVIOUS ERROR WAS DETECTED ON LIN
PARTIAL PARSE TO THIS POINT IS:  <BLOCK END>
  22 | 1|  69 |EOF

*** ERROR, EOF MISSING OR COMMENT STARTING IN COLUMN 1..  LAST PREVIOUS ERROR WA
  22 | 1|  69 | ;END EOF
RESUME:    |
*** ERROR, ILLEGAL SYMBOL PAIR: <BEGIN> END.  LAST PREVIOUS ERROR WAS DETECTED O
PARTIAL PARSE TO THIS POINT IS:  <BEGIN>
```

Figure 1 (Continued)

```
  R I C A L G O L - 2  C O M P I L A T I O N              VERSION OF DECEMBER 4,
TODAY IS DECEMBER 4, 1971.  CLOCK TIME = 15:54:48.53.

CARD |BL| SYL |
   1 |  0|   0 |BEGIN LOCAL I,J,K;
   2 |  1|   0 |K:=2;
   3 |  1|   5 |I:=J:=K; WRITE(K,J,I);
   4 |  1|  24 |IF (I LEQ 2) THEN I:=5;
   5 |  1|  37 |K:=J:=I; WRITE(I,J,K);
   6 |  1|  56 | BEGIN LOCAL LYLE,JERRY,MARY,JOHN,JIM,HEINRICH,KARL,KATHY;
   7 |  2|  56 |   BEGIM LOCEL RICH;
**** ERROR **** MISPELLED RESERVED WORD  BEGIM  REPLACED BY       BEGIN
**** ERROR **** MISPELLED RESERVED WORD  LOCEL  REPLACED BY       LOCAL
   8 |  3|  56 |   FOX LILE:=1 STEP 2 UNTIL 2 DO JAM:=1;
**** ERROR **** MISSPELLED RESERVE WORD FOX REPLACED   BY  FOR
**** ERROR **** MISSPELLED IDENTIFIER  LILE  REPLACED BY  LYLE
**** ERROR **** MISSPELLED IDENTIFIER  JAM  REPLACED BY  JIM
   9 |  3| 109 |   FOR LYLL:=1 STAP 2 UNTIL 2 DO KERL:=2;
**** ERROR **** MISSPELLED RESERVE WORD STAP REPLACED   BY  STEP
**** ERROR **** MISSPELLED IDENTIFIER  LYLL  REPLACED BY  LYLE
**** ERROR **** MISSPELLED IDENTIFIER  KERL  REPLACED BY  KARL
  10 |  3| 162 |   FOR JARRY:=1 STEP 2 UNTLI 2 DO KETHY:=3;
**** ERROR **** MISSPELLED IDENTIFIER  JARRY  REPLACED BY  JERRY
**** ERROR **** MISSPELLED RESERVE WORD UNTLI REPLACED   BY  UNTIL
**** ERROR **** MISSPELLED IDENTIFIER  KETHY  REPLACED BY  KATHY
  11 |  3| 216 |   MARY:=3;
  12 |  3| 221 |   WRITA(MARY);
**** ERROR **** MISSPELLED RESERVE WORD WRITA REPLACED   BY  WRITE
  13 |  3| 225 |   IF (MERY LST 4) THAN MARY:=9;
**** ERROR **** MISSPELLED RESERVE WORD LST REPLACED   BY  LSS
**** ERROR **** MISSPELLED IDENTIFIER  MERY  REPLACED BY  MARY
**** ERROR **** MISSPELLED RESERVE WORD THAN REPLACED   BY  THEN
  14 |  3| 238 |   WRITE(MARY);
  15 |  3| 242 |   HEINRICH:=2;
  16 |  3| 247 |   CASS HEINRICK FO
**** ERROR **** MISSPELLED RESERVE WORD CASS REPLACED   BY  CASE
**** ERROR **** MISSPELLED RESERVE WORD FO REPLACED   BY  OF
**** ERROR **** MISSPELLED IDENTIFIER  HEINRICK  REPLACED BY  HEINRICH
  17 |  3| 249 |     BEGIN WRITE(HEINRICK); WRITE(HEINRICK);
**** ERROR **** MISSPELLED IDENTIFIER  HEINRICK  REPLACED BY  HEINRICH
**** ERROR **** MISSPELLED IDENTIFIER  HEINRICK  REPLACED BY  HEINRICH
  18 |  5| 290 |           WRITE(HIENRICH); END;
**** ERROR **** MISSPELLED IDENTIFIER  HIENRICH  REPLACED BY  HEINRICH
  19 |  3| 317 |   EMD;
**** ERROR **** MISSPELLED RESERVE WORD EMD REPLACED   BY  END
  20 |  2| 317 |END
```

Figure 2. Test Program – BALGOL Compiler

```
   21 |  2|  317 |END
   22 |  1|  317 |EOF
PRT=12, DATA=16, CODE=80 (WORDS).
CODE FILE WRITTEN
END OF COMPILATION DECEMBER 4, 1971.  CLOCK TIME = 15:54:49.97.

22 CARDS WERE READ.
NO ERRORS WERE DETECTED.
0
SET UP TIME                 0:0:0.10.
ACTUAL COMPILATION TIME     0:0:1.34.
CLEAN-UP TIME AT END        0:0:0.00.
TOTAL TIME IN COMPILER      0:0:1.44.
COMPILATION RATE :985 CARDS PER MINUTE.



I  TYPE IS  SIMPLE  BLOCK IS  1  DECLARED AT LINE  1
I  WAS REFERENCED  6  TIMES ON THE FOLLOWING LINES
3
VALUE CHANGED ON LINE  3
3
4
4
VALUE CHANGED ON LINE  4
5
5


J  TYPE IS  SIMPLE  BLOCK IS  1  DECLARED AT LINE  1
J  WAS REFERENCED  4  TIMES ON THE FOLLOWING LINES
3
3
5
5


K  TYPE IS  SIMPLE  BLOCK IS  1  DECLARED AT LINE  1
K  WAS REFERENCED  5  TIMES ON THE FOLLOWING LINES
2
VALUE CHANGED ON LINE  2
3
3
5
VALUE CHANGED ON LINE  5
5
```

Figure 2 (Continued)

```
LYLE  TYPE IS  SIMPLE  BLOCK IS  2  DECLARED AT LINE  6
LYLE  WAS REFERENCED  2  TIMES ON THE FOLLOWING LINES
8
9


JERRY  TYPE IS  SIMPLE  BLOCK IS  2  DECLARED AT LINE  6
JERRY  WAS REFERENCED  1  TIMES ON THE FOLLOWING LINES
10


MARY  TYPE IS  SIMPLE  BLOCK IS  2  DECLARED AT LINE  6
MARY  WAS REFERENCED  5  TIMES ON THE FOLLOWING LINES
11
VALUE CHANGED ON LINE  11
12
13
13
VALUE CHANGED ON LINE  13
14


JOHN  TYPE IS  SIMPLE  BLOCK IS  2  DECLARED AT LINE  6
JOHN  WAS REFERENCED  0  TIMES ON THE FOLLOWING LINES


JIM  TYPE IS  SIMPLE  BLOCK IS  2  DECLARED AT LINE  6
JIM  WAS REFERENCED  1  TIMES ON THE FOLLOWING LINES
8


HEINRICH  TYPE IS  SIMPLE  BLOCK IS  2  DECLARED AT LINE  6
HEINRICH  WAS REFERENCED  5  TIMES ON THE FOLLOWING LINES
15
VALUE CHANGED ON LINE  15
16
17
17
18


KARL  TYPE IS  SIMPLE  BLOCK IS  2  DECLARED AT LINE  6
KARL  WAS REFERENCED  1  TIMES ON THE FOLLOWING LINES
9


KATHY  TYPE IS  SIMPLE  BLOCK IS  2  DECLARED AT LINE  6
```

Figure 2 (Continued)

```
KATHY  WAS REFERENCED  1  TIMES ON THE FOLLOWING LINES
10

RICH  TYPE IS  SIMPLE  BLOCK IS  3  DECLARED AT LINE  7
RICH  WAS REFERENCED  0  TIMES ON THE FOLLOWING LINES


PRT=12, DATA=16, CODE=80
MEMORY INITIALIZED, STARTING AT 308, ENDING AT 8191

2               2               2
5               5               5
3
9
2
```

Figure 2 (Continued)

II. SPELLING CORRECTION

A. EARLY WORK

One of the most obvious beginning points for the detection and correction of errors in compilers is the correction of spelling errors in both user defined identifiers and reserved words. This beginning point is natural since one of the most common situations to arise is ⟨IDENTIFIER⟩⟨IDENTIFIER⟩ , an "ILLEGAL SYMBOL PAIR". Spelling correction is clearly related to some of the initial work in the area of pattern matching done in the late 50's and early 60's. Works in this area include those of Cyril Albegra on string similarity and misspelling and Fred Damerau's work on computer detection and correction of spelling errors.

The two major efforts in the area were: 1) Charles Blair's program for correcting spelling errors, and 2) Freeman's PhD dissertation at Cornell, in which he worked with error correction in CORC, Cornell Computing language.

1. Blair's Work

Blair's work [2] was developed as a simple, heuristic procedure for the correction of simple spelling errors. The program uses a dictionary of correct spellings and the idea of similarity to match misspellings with their corresponding dictionary word.

The algorithm works on the basis of computing an abbreviation for all the dictionary words and then computes an abbreviation for any misspelling and bases its word matching on the abbreviations. The idea of the abbreviation is to isolate the "kernel" of the word, the most important letters which uniquely describe the word.

The abbreviation is computed on the basis of importance values assigned to the various letters of the alphabet and a position factor assigned to the various positions in the word. Each letter of the word is assigned a value by summing these two factors. The abbreviation is the "n" letters of the word with the lowest value, where "n" is the length of the abbreviation desired. The following is an example of length 4.

A	B	S	O	R	B	E	N	T		A	B	S	O	R	B	A	N	T
5	1	5	4	4	1	7	3	3	Letter score	5	1	5	4	4	1	5	3	3
0	2	4	5	5	5	4	3	1	Position score	0	2	4	5	5	5	4	3	1
5	3	9	9	9	6	11	6	4	Sum	5	3	9	9	9	6	9	6	4
	*	*	*		*	*			Deletions			*	*	*		*	*	
			A	B	B	T			Abbreviation	A	B	B	T					

The results are impressive. With an abbreviation length of 4 the program recognized 89 of 117, and only totally failed on 2 words, but the author implies an abbreviation length of 5 would have solved the problem. The program has only two major problems 1) examination of words not in the dictionary, and 2) gross misspellings.

Blair designed his program specifically to correct errors made by humans, i.e. errors produced by humans in such activities as key-punching and not to correct random errors such as transmission failures. The program is less effective in correcting this type of error.

2. Freeman's Work

Freeman's method is a complex system based on the probability that one identifier is a misspelling of another. Freeman's work involves the use of a pseudo scoring polynomial, a technique very similar to the artificial intelligence work with polynomial evaluation. The scoring polynomial involves a variety of information including: 1) the number

of letters in the suspicious identifier which match an identifier in the symbol table, 2) the number of letters which match after the transpositions and substitutions have been made, and 3) the number of letters which match after common keypunch errors are considered. This system is capable of detecting and correcting a great many errors but its space and time requirements are correspondingly large.

3. Morgan's Method

Morgan suggests a less powerful method, which he claims would be able to correct approximately 80 per cent of all spelling errors. This method considers only four cases: 1) one wrong letter, 2) one letter missing, 3) one extra character added, and 4) two adjacent letters transposed.

B. CHOOSING AN ALGORITHM

In choosing an algorithm for implementation three factors were considered: 1) the complexity of coding the algorithm into XPL, 2) the time required to perform the test on an undeclared identifier, and 3) the space requirements of the algorithm. The latter consideration is the result of large compiler overhead and the fact that most priority systems favor smaller jobs.

In considering the first factor, one reason for the rejection of complicated methods such as Freeman's algorithm was the lack of real arithmetic in XPL. XPL does have good string processing and comparing facilities which add greatly to the ease of implementing Morgan's algorithm. The second factor, speed, was considered because of the nature of compilers and their use. Compilers which have large use by students need more extensive error correction such as provided by Freeman's algorithm, while production compilers need far less correction

capabilities. The final requirement was space in implementing the various algorithms. This led to conclusions similar to those of complexity.

For the purpose of this experimental compiler Morgan's algorithm proved to be easily implementable and quite effective. A further consideration was the claim by proponents of the algorithm that it could correct eighty per cent of all spelling errors and this was an acceptable level of correction.

C. DETECTION OF MISSPELLINGS

Detection of spelling errors is a complex problem. The problem is compounded by the various conventions the different languages have. Languages fall into two main classes with respect to identifiers: 1) those with all identifiers predeclared before use, and 2) those in which a new simple variable may be declared simply by use whenever needed. Other aspects which complicate the picture are: 1) the nature of identifiers, i.e. attributes — simple, array or procedure, 2) the varying types, e.g. integer, real, etc., and 3) the fact that misspelled reserved words become identifiers. Virtually all languages require the predeclaration of arrays. The problem of type is solved by FORTRAN-like languages by their type conventions, and ALGOL-like languages require predeclaration.

Different systems have different situations which suggest that an identifier might be a misspelling. For example, in FORTRAN — FORMET (***) would obviously be suspect, and in ALGOL — FOR ⟨assignment⟩ UNTL would be certainly suspect. From the compiler's viewpoint these situations show up as: 1) An identifier appears when a reserve word is expected; 2) An

identifier is used as a type other than its declared type; 3) An identifier appears only on the right side of assignment statements; and 4) An identifier which should have been predeclared does not appear in the symbol table. Each of the situations is unique and requires a special approach to solve it.

At times during the parsing of input text a reserve word is expected, e.g. IF <boolean expression> THEN , and if the identifier THAN occurs it can be assumed with reasonable accuracy that THEN is needed.

Type declarations also yield an obvious opportunity for detecting misspelled identifiers, e.g. if a label TOP is declared and the label TAP occurs the switch to TOP would be an obvious move.

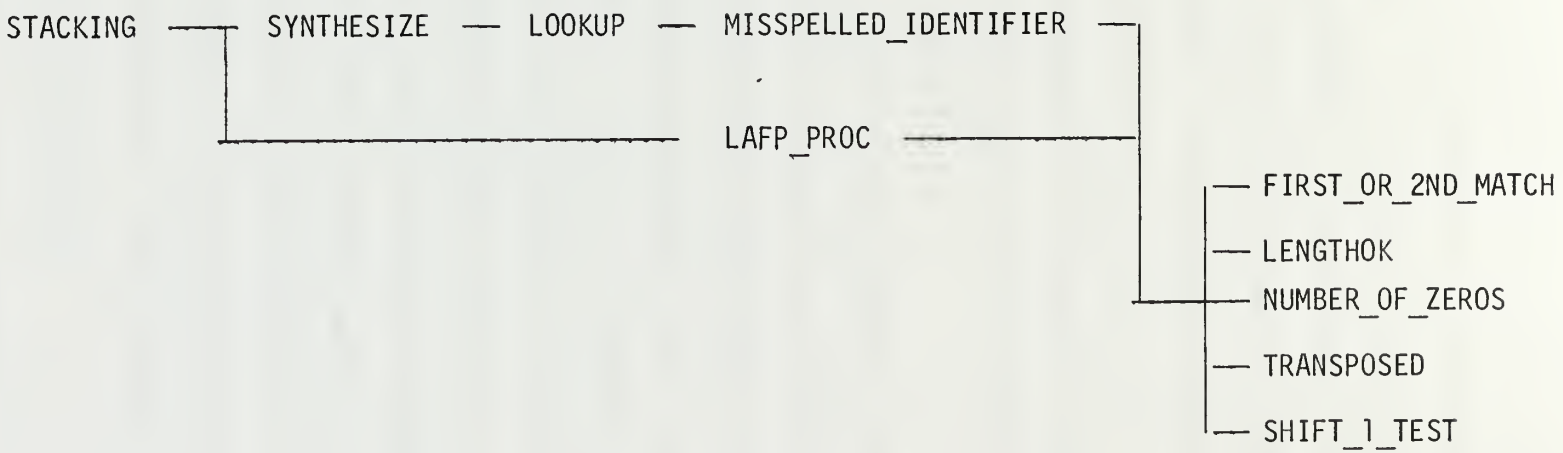
Single occurrence of a variable can lead to suspicion. If the variable ROW1 occurs at several places in the program but in one equation ROWU occurs on the right hand side of an <assignment statement> but never occurs on the left of an assignment it is likely that a keypunching error has occurred and should be replaced by ROW1.

When all variables are predeclared and an integer KOUNT is referenced but does not appear in the symbol table as an integer or as any other type but an integer COUNT does appear the replacement seems to be the most logical solution to the problem.

The scheme presented in this thesis uses the first, second and fourth methods as BALGOL requires all predeclaration. The first method is the one primarily detected by "ILLEGAL SYMBOL PAIRS".

D. SPELLING CORRECTION ALGORITHM

The two main procedures (see figure 3) are MISSPELLED_IDENTIFIER and LAFP_PROC. MISSPELLED_IDENTIFIER is the main procedure used in the detection of misspellings. MISSPELLED_IDENTIFIER is called every time



Misspelled Identifier Correction Algorithm

Figure 3

a storage location for an identifier is requested and no match in the symbol table is found. The compiler used is the BALGOL-2 compiler which is ALGOL-like with all identifiers predeclared. MISSPELLED_IDENTIFIER is also called from stacking when the error "ILLEGAL SYMBOL PAIR" is encountered. This procedure operates by a series of calls to the five working procedures. (See figure 3) The procedure has two basic parts, the first used for misspelled identifiers and the second for misspelled reserve words. This procedure works only if the length of the questionable identifier is greater than one. The symbol table is searched top-down because of the heuristic decision that the most recently declared identifier is the most likely to be used and should be the first matched if possible.

The second calling procedure is LAFP_PROC which is a special purpose procedure used only when one of the special reserve words, LOCAL, ARRAY, FUNCTION, PROCEDURE, BEGIN or END is expected. LAFP is called from STACKING and is called only after the detection of the head symbol BEGIN. Once a BEGIN is detected an identifier in token will be stacked automatically, but if the identifier is really a reserve word of this special class then great damage is done, e.g. BEGIN LOCEL A would cause the identifier A not to be declared properly. This damage would have to be corrected if this situation were not handled specially, e.g. if LOCAL is misspelled then an extra reduction is made and parsing fails. In order to avoid removal of code which has already been generated, a test is made to all occurrences of BEGIN followed by an identifier.

E. ACTUAL IMPLEMENTATION

Spelling correction involves the selective calling of MISSPELLED_ IDENTIFIER and LAFP_PROC when a suspicious identifier is encountered or

when an "ILLEGAL SYMBOL PAIR" is detected. Suspicious identifiers are detected when the PRT address of an identifier is needed and the identifier does not occur in the symbol table. This is detected in the procedure LOOKUP, which is designed to locate identifiers in the symbol table and return their PRT address. BALGOL-2 is a blockstructured language with integers, integer arrays, functions and procedures only, and requires the declarations of all identifiers before use. No labels are allowed, thus assuring the detection of misspellings by reaching symbol table entry 0 without a match.

Misspelled identifiers are corrected with relative ease. The error message "**** ERROR **** MISSPELLED IDENTIFIER __ REPLACED BY __" is emitted. The PRT address of the symbol table entry matched is then returned as the address of the misspelled identifier.

Misspelled reserve words are more difficult to detect, since they take on the appearance of identifiers and often get to the parse stack before being detected. The nature of reserved words allows them to take on the appearance of many other legal combinations not covered by LAFP_PROC, e.g. WRITE (A,B); has the same basic structure as PROCED (A,B); where PROCED is the invocation of a procedure. Thus the spelling of WRIT (A,B) could cause erroneous parsing by allowing an identifier to appear on the stack in place of the reserve word WRITE. Misspelled reserve words can also cause illegal symbol pairs, e.g. if WHILE A LSS B were to appear as WHIL A LSS B then the occurrence of two identifiers together would cause the detection of the misspelling.

The problem of misspelled identifiers could be attacked by a call to LOOKUP from the procedure SCAN whenever an identifier is detected on the input stream. This method has one basic drawback, the time required to look up every identifier.

Actual correction of misspelled reserve words not covered in LAFP_PROC is performed in the procedure STACKING which determines how many input symbols are to be placed on the stack before attempting a reduction.

Stacking makes its decision from the top of the parse stack and the symbol in token. If an illegal symbol pair is detected then one attempt is made to resolve the conflict, providing either the top element of the parse stack or token is an identifier. This is done by checking the top of the stack first for misspellings and correcting through the use of a case statement on all the reserve words. If this fails the token value is then checked and corrected in the same manner. If both fail other methods must be employed.

F. OTHER CONSIDERATIONS FOR SPELLING CORRECTION

In general there are two possible considerations which directly affect the ease of detecting misspellings. The first is already implemented in the BALGOL-2 compiler, i.e. predeclaration of all variables by type. This would allow easy detection of misspellings and facilitate correction. The second consideration comes when writing the BNF for the language, namely never allow <identifier><identifier> or <identifier> terminal to be legal if reserve word terminal is legal, e.g. instead of allowing <identifier> "(" to be a procedure invocation or the misspelling of READ, WRITE, IF, WHILE or CASE. This could be accomplished by enclosing boolean expressions for an IF, WHILE or CASE statement in vertical bars rather than "(", using broken brackets, <,> , around array subscripts, and using exclamation points to delimit lists of input/output statements. This would allow misspellings to be detected as illegal

symbol pairs rather than after several reductions have been made. Although this method would make error correction easier it might lead to additional errors due to the large symbol set.

III. ERRORS OF INSERTION/DELETION

A. INITIAL COMMENTS

The basis for the correction of errors of insertion or deletion is a data base which provides information about the local context at the point where the error was detected. In this system the context consists of the relationships between terminals.

B. PREPARATION OF A DATA BASE

In preparing a data base for the compiler's use, several considerations were studied; the first consideration was what information was available from the tables produced by the XPL analyzer; second was the consideration of how the XPL compiler presently detects errors; third was the meaning of these errors in terms of code generation; and the final consideration was size requirements and access time of the data base.

The XPL analyzer produces a great volume of information, including a vocabulary list and a C1 stacking decision array. It is around this array that all parsing in the compiler revolves. This C1 array contains one position for each element in the set {NS X NT}, NS is the number of symbols in the vocabulary, NT is the number of terminals, the value of which is 0,1,2,3 which indicates the stacking decision for the compiler. Thus the first method of error recognition is the occurrence of two symbols, X and Y, where the position (X,Y) of C1 is equal to zero. This also leads to a starting point for error correction. These errors may be correctible by insertion or deletion of proper symbols.

The XPL analyzer also produces an array HDTB, PRDTB, PRTB which holds the productions of the grammar and are used in making the reduction in the parsing of statements. These arrays along with the context arrays CONTEXT_CASE, CONTEXT_TRIPLE, and LEFT_CONTEXT are used to decide at a given stage if any reduction can be made and if so which production should be made. This yields a second type of error, the "NO PRODUCTION APPLICABLE." This type of error occurs when terminals, which make sense in a stacking decision, appear together and are stacked. These errors are far more complicated. Their correction would require a compiler to reverse its parser, wipe out code which it had already generated, make an evaluation of the error and alter the input stream before reparsing the erroneous input. The "ILLEGAL SYMBOL PAIR" error once detected may require only the insertion of a symbol between the present stack top and token symbols, e.g. <identifier><identifier> could be corrected by the insertion of a "+", or as in the case of <number><number>, e.g. STEP 1 1 UNTIL, could be corrected by the deletion of the second one. This technique doesn't guarantee results but it could lead in many cases to the continuation of the parsing and the possible proper execution of the program.

The final consideration is the space required and access time of the data base. Obviously too much space or too much time would be detrimental to such a system.

There are four basic considerations in developing a data base for compiler correction of errors: 1) available information, 2) present error detection facilities, 3) meaning of these errors in terms of correction, and 4) space limitations. All these considerations must be taken into account when developing a base of information to be used when correcting errors.

C. INTRODUCTION TO THE ANALYZER

Using the tables produced by the XPL analyzer three inputs are prepared for the new syntax analyzer produced as part of this paper. The first input is a simple list of the vocabulary for the grammar, all terminals followed by the nonterminals. The second is the list of productions for the grammar which is in the same form as for the XPL analyzer. Finally the table of pseudo legal triples, which is produced by an XPL program is prepared. A pseudo legal triple is defined to be a string of three terminals, A B C, such that the pair A C is an illegal symbol pair and further a stacking decision exists between A and B and B and C, e.g. ⟨identifier⟩ + ⟨identifier⟩ is actually a good triple while ";" ⟨identifier⟩")" is a pseudo legal triple but cannot be generated from the grammar. The legal triples among these pseudo triples will give valuable information on the possible insertions between terminals.

The basis for the analyzer is a suggestion by David Gries [5] in his book Compiler Construction for Digital Computers. Gries suggests four possible methods of recovering from a given situation (x1,T).

1. If there exists a rule U::=x1 z T ... in the grammar, a terminal string q should be inserted such that z $\rightarrow^*$ q.
2. If there exists a rule U::=x1V ... such that V $\rightarrow$zT ..., a terminal string q should be inserted such that z $\rightarrow^*$ t.
3. If there exists a rule U::= ... VT .. such that V $\rightarrow^+$ ... Wz2 and W::= x1z1 is a rule, a terminal string q should be inserted such that z = z1z2 $\rightarrow^*$ q.
4. If none of the above apply, T should be deleted.

D. THE ANALYZER

The syntax analyzer produces a large amount of information which is useful in the analysis of the problem of insertions/deletions, but the

basic output is the list of legal triples which were extracted from the list of pseudo legal triples. The other major outputs are the right sets and right pair sets of all the nonterminals as well as all legal triples. The right sets and right pair sets of a terminal are the sets of terminals generated at the far right of a generation from that nonterminal. Each output builds on the previous output; the right sets are used to generate the right pair sets and both are used to generate the legal triples.

A procedure which is typical of the recursive procedures on which the analyzer is based is BRS. This procedure is given in figure 4, with a flow chart shown in figure 5. The procedure's one calling argument is an element of the vocabulary.

The basic aim of the analyzer is to provide output which the compiler writer, and hopefully later a more complicated analyzer, can use heuristically to provide further input to the compiler tables. This input would provide: 1) information to the compiler when it is no longer able to continue the parsing, 2) information on possible insertions which could be used to continue the parse and hopefully in some cases provide a correct solution, 3) deletion input by default, e.g. if no insertion is legally possible then delete the value in token.

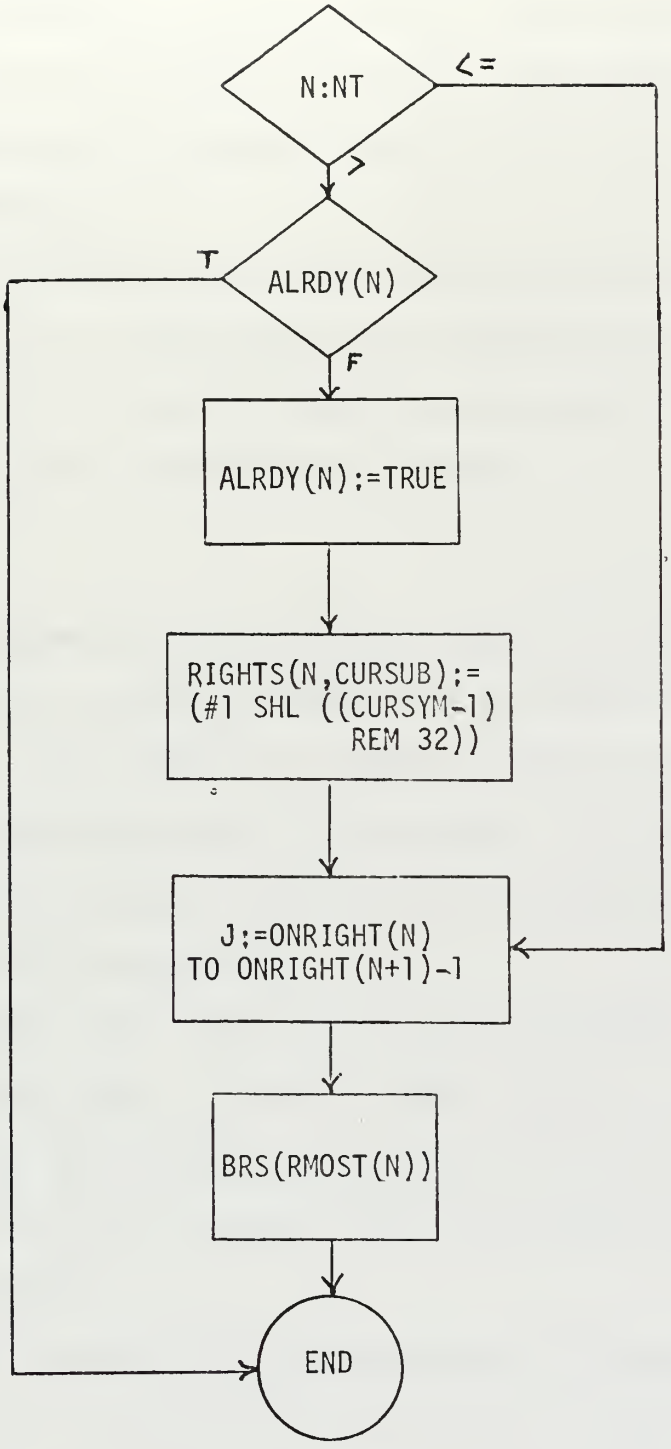
E. HEURISTICS

Although the analyzer was able to cut the number of pseudo legal triples for the BALGOL test grammar in half there were still over 1500 legal triples remaining. The basic heuristics used to eliminate legal triples, after the analyzer has removed all nonlegal triples, must be such that they eliminate all duplication of equivalent terminals. Equivalent terminals are two terminals which have the same roles,

```
PROCEDURE BRS(INTEGER VALUE N);
   BEGIN
      IF N>NT THEN
         IF ALRDY(N) THEN GO TO RETN
                     ELSE BEGIN ALRDY(N) := TRUE;
                                RIGHTS(N,CURSUB) := RIGHTS(N,CURSUB) OR
                                   (#1 SHL ((CURSYMBOL-1) REM 32));
                          END;
      FOR J:= ONRIGHT(N) UNTIL ONRIGHT(N+1)-1 DO  BRS(RMOST(J));
    RETN:  END BRS;
```

BRS FROM ANALYZER

Figure 4



FLOW CHART — BRS

Figure 5

i.e. produce the same parse or essentially the same parse when inserted between two terminals, e.g. in most grammars, identifiers and numbers play the same role in arithmetic expressions. These equivalent terminals should be generated by an algorithm similar to the one which generates singles of nonterminals.

The basis for the deletion of legal triples must be the elimination of all triples which are equivalent in an insertion set. The insertion set of two terminals, A and B, is defined as the set of all terminals which could be legally inserted between A and B.

F. RESULTS

Two large grammars were run with the analyzer, the BALGOL-2 grammar used in the spelling correction compiler and the ALGOL-E grammar written by Gary Kildall. Both grammars are similar in terms of their basic definitions. The BALGOL grammar has 142 symbols in the vocabulary, 56 terminals and 157 productions, while the ALGOL-E grammar has 121 symbols, 49 terminals and 132 productions.

When the pseudo legal triples were produced for BALGOL there were 2995 pseudo legal triples. ALGOL-E had 2396. After being processed by the analyzer only 1519 triples remained for BALGOL and only 1308 for ALGOL-E. This reduction is roughly half, 50.3% legal for BALGOL and 54.6% legal in ALGOL-E.

Now consider the heuristics that the compiler writer generates to reduce the number of triples which will actually be added to the compiler tables. The triples would then be consulted by an error subroutine to add or delete terminals from the input text to allow the parsing to continue.

The following are the simple heuristics which were applied to the legal triples outputed by the analyzer.

1. If the operators =,-,*,/, '(exponentiation) are all legal insertions, use + and eliminate the others.
2. If the logical operators AND and OR are both legal use OR and delete AND.
3. If the logical comparatives LSS, LEQ, EQL, NEQ, GEQ, and GTR are all possible insertions use LSS and delete all others.
4. If any subset of these operators or comparatives appears in an insertion set choose one and delete all others.
5. The subset {<identifier>, <number>} is reduced to the set {<number>} with the insertion of 1 following the operator's *, / or ' and 0 elsewhere.
6. Wherever the set {<procedure invocation> , <simple procedure invocation>} appear in an insertion set delete <procedure invocation> and insert TRACE as the terminal.
7. The set {<operation> , <data descriptor>} is reduced to the set {<operation>} with a trace being generated.
8. The set {<identifier> , <string>} is reduced to {<string>} with "ERROR INSERTION" being the insertion.
9. The set {<identifier>} is totally deleted with a special procedure to handle this case.

After applying these nine heuristics to BALGOL the insertion triples list was reduced by 460 to 1059, a reduction of 30.3% while in ALGOL-E the reduction was 490 to 818, a reduction of 37.6%. BALGOL contains 713 insertion sets and ALGOL-E has 566. Of these 713 originally only 221 had unique insertions while after the use of the heuristics this

number grew to 509, i.e. 71% of the insertion sets have a unique insertion. ALGOL-E showed a similar increase with 173 initially and 414 unique insertions after the application of the heuristics, i.e. 73% have only one insertion possible. These reductions can best be seen in figure 6, a table showing the relative numbers of the insertion sets before and after use of the heuristics. Thus of the original 2996 pseudo legal triples only 1059 triples remain to be considered, a reduction of 64.8%. ALGOL-E originally had 2396 triples. However, only 818 remain, a reduction of 65.9%.

In developing the heuristics used to accomplish the above reductions the grammar and insertion sets were examined to determine which terminals played equivalent roles in the productions, e.g. <identifier>, <number> in arithmetic expressions. When equivalent terminals were detected all but one was eliminated from the insertion set and other similar insertion sets were also examined. The equivalent terminal to be retained was determined as follows: 1) if the terminals were simple terminals, e.g. =,-,(etc., simple choice was used; or 2) if the terminals had attributes, e.g. <number>, a value, <identifier>, a character representation, the simplest attribute to determine was used, e.g. <number>, <identifier> the <number> was chosen since the attribute 1 could always be assigned to it. These equivalent terminals were initially determined by consulting the singles table, a single of a nonterminal x is defined to be a terminal t such that x —> *t. These methods seem to be such that they could be included in an analyzer and used to reduce all insertion sets to a single element. The terminal would be used every time the situation occurred. If it was successful the parse would continue, if not other actions would have to be taken.

NUMBER OF INSERTION SETS CONTAINING N ELEMENTS

N	BALGOL BEFORE	AFTER	ALGOL-E BEFORE	AFTER
0	0	53	0	48
1	221	509	173	414
2	398	73	326	51
3	25	14	18	7
4	44	39	27	24
5	5	5	5	5
6	1	1	2	2
7	4	4	0	0
8	0	0	0	0
9	0	10	0	9
10	4	0	0	0
11	0	4	0	5
12	0	0	0	1
13	0	1	0	0
14	6	0	0	0
15	0	0	3	0
16	4	0	0	0
17	0	0	1	0
18	1	0	0	0
19	0	0	6	0
20	0	0	0	0
21	0	0	4	0
22	0	0	1	0
	713	713	566	566

Table of insertion sets

figure 6

IV. CONCLUSION

Figure 1 is a prime example of the minor errors which cause needless resubmission because the program contains only minor spelling errors. These errors are an area of major frustration to new users and to students who are unfamiliar with keypunches.

The problem of misspelled identifiers is covered in this paper as well as some initial work on the problem of errors of insertion/deletion. The problem of missing terminals in the input text is a very complicated one. The compiler must first generate any logical insertion, and if more than one insertion is possible, it must make a decision as to which terminal to use, when to abandon the path and try again. The basic analyzer to produce a first input is presented here. This analyzer could be the basis for a future more complete error correcting compiler.

I feel the compiler produced could be a model for the procedures which could be inserted into any XPL based compiler with predeclaration and could easily be modified for FORTRAN-like compilers. I feel such routines should be basic to any nonproduction compiler or compiler used primarily by students. I believe the analyzer is a good first step toward the solving of the problem of implementation of Gries' suggestions and toward the basic aim of compilers which can cope with almost any problem they might encounter.

V. TOPICS FOR FURTHER CONSIDERATION

A. DATA BASE IMPLEMENTATION

The most obvious extension would be the implementation of this data base concept into the XPL system. This would involve outputing of numerical data from the analyzer into another interface program to produce table input acceptable to XPL. The data base might be built as follows. There could be two index matrices of dimension NT which index into the data base by the token and top of the stack values, e.g. TOKEN_INDEX would locate the base of the section containing information about the token and STACK_INDEX would locate the subsection associated with the top element of the stack. The basic data base could be stored in an integer (fixed) array with the 0^{th} byte empty, 1^{st} byte containing the stack element, 2^{nd} byte containing the insertion and 3^{rd} byte containing the token value.

Actual implementation in the compiler would not be an easy matter since whenever making an insertion with more than one alternative the entire context must be saved. Once the insertion has been made a method of determining its correctness must be found. If the insertion is not successful then a method must be devised to restore the system to its original state. Finally if no insertion leads to a successful continuation of the parse then a more effective method must be found. A possible method might be the cleaning of the stack back to the last ";", inserting a statement and then discarding text until the next ";" is encountered.

B. EXAMINE MORE CONTEXT

One method of making the analyzer more effective in identifying unique insertions and building a data base for correction of insertion/deletion errors would be to expand the analyzer to look at more local context, perhaps quadruples in place of triples.

C. MULTIPLE INSERTIONS/DELETIONS

Presently the analyzer is set to develop only single insertions but in some cases several symbols may need to be inserted or deleted in order to allow parsing to continue properly. An analyzer might be developed to aid in the solution of this problem.

D. SPECIAL GRAMMARS

One area of interest which has merit for further studies is the area of special error correcting grammars which give a maximum of local context at all times. These grammars should eliminate as many multiple insertions as possible.

E. HEURISTICS TO ELIMINATE TRIPLES

Further development could well be done with the heuristics used to eliminate triples. These heuristics might also be added in some way to the analyzer.

F. MORE POWERFUL SPELLING CORRECTION

The spelling correction might be made as powerful as desired, with Freeman's method being, perhaps, the most powerful. The present routines could be made more powerful by the insertion of a routine to handle normal upper/lower case spelling errors with more guaranteed results.

```
DECLARE NSY LITERALLY '121',NT LITERALLY '49';
DECLARE V(NSY) CHARACTER INITIAL ( '<ERROR: TOKEN = 0>', ';', ',', ':', '=',
   '<>', '+', '-', '*', '/', '(', ')', 'OR', 'OF', 'IF', 'DO',
   '_', '.', 'END', 'NOT', 'AND', 'LSS', 'LEQ', 'EQL', 'NEQ', 'GEQ', 'GTR',
   'FOR', 'TAB', 'ELSE', 'CASE', 'THEN', 'STEP', 'READ', 'BEGIN', 'LOCAL',
   'ARRAY', 'WHILE', 'UNTIL', 'PROCEDURE', 'FORWARD', 'WRITE', 'NUMBER>',
   'FUNCTION', '<STRING>', '<IDENTIFIER>',
   '<PROCEDURE INVOCATION>', '<SIMPLE PROCEDURE INVOCATION>', '<TERM>',
   '<STEP>', '<BLOCK>', '<BEGIN>', '<WHILE>', '<PROGRAM>', '<PRIMARY>',
   '<VARIABLE>', '<RELATION>', '<IF GROUP>', '<BLOCK END>', '<STATEMENT>',
   '<IF CLAUSE>', '<READ HEAD>', '<BLOCK HEAD>', '<BLOCK BODY>',
   '<LOCAL HEAD>', '<RIGHT PART>', '<EXPRESSION>', '<ARRAY LIST>',
   '<ARRAY HEAD>', '<BOUND PAIR>', '<FOR CLAUSE>', '<WRITE HEAD>',
   '<DECLARATION>', '<LOWER BOUND>', '<PROC&PARAMS>', '<IF STATEMENT>',
   '<CALL HEADING>', '<BOOLEAN TERM>', '<WHILE CLAUSE>', '<DO STATEMENT>',
   '<CASE HEADING>', '<UNTIL CLAUSE>', '<FOR STATEMENT>', '<IF EXPRESSION>',
   '<FUNCTION HEAD>', '<IF ELSE GROUP>', '<READ STATEMENT>',
   '<CASE STATEMENT>', '<SUBSCRIPT HEAD>', '<PROCEDURE CALL>',
   '<PARAMLESS PROC>', '<PROCEDURE HEAD>', '<TAB EXPRESSION>',
   '<DECLARATION SET>', '<WHILE STATEMENT>', '<WRITE STATEMENT>',
   '<PRIMARY ELEMENT>', '<BOOLEAN PRIMARY>', '<BOUND PAIR LIST>',
   '<BOUND PAIR HEAD>', '<FUNCTION&PARAMS>', '<STEP EXPRESSION>',
   '<ARRAY DECLARATION>', '<PROCEDURE HEADING>', '<FORWARD PROCEDURE>',
   '<FORWARD FUNC HEAD>', '<FORWARD PROC HEAD>', '<SIMPLE DECLARATION>',
   '<BOOLEAN EXPRESSION>', '<LOGICAL EXPRESSION>', '<SUBPROGRAM HEADING>',
   '<PARAMLESS FUNCTION>', '<FORWARD DECLARATION>', '<ASSIGNMENT STATEMENT>',
   '<SUBSCRIPTED VARIABLE>', '<ARITHMETIC EXPRESSION>',
   '<SUBPROGRAM DECLARATION>');
DECLARE V_INDEX(29) BIT(8) INITIAL ( 1, 14, 18, 30, 35, 41, 43, 46, 47,
   47, 47, 48, 48, 48, 48, 48, 48, 48, 49, 49, 49, 49, 49,
   49, 50);
DECLARE C1(NSY) BIT(100) INITIAL (
   "(2) 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000 00000",
   "(2) 00000 00000 03001 00000 00030 03003 31130 31301 01333",
   "(2) 00000 00220 00200 02000 00002 00000 00020 00020 20200",
   "(2) 00001 00110 00100 01000 00000 00000 00010 00010 00100",
   "(2) 00001 00110 00100 01000 00000 00000 00010 00010 00100",
   "(2) 00000 20220 20200 02002 02222 22000 00002 00020 00200",
   "(2) 00000 20220 20200 02002 02222 22000 00002 00020 00200",
   "(2) 00000 00000 00000 00000 00000 00000 00000 00010 00100",
   "(2) 00000 00000 00100 01000 00000 00000 00000 00010 00100",
   "(2) 00000 00000 00100 01000 00000 00000 00000 00010 00100",
   "(2) 00000 00000 00100 01000 00000 00000 00000 00010 00100",
   "(2) 00000 00330 00300 03000 10000 00002 00000 00030 20300",
   "(2) 00220 20222 20022 20202 02222 22000 20220 00002 00000",
   "(2) 00000 00110 00100 01000 10000 00000 00000 00010 00100",
```

```
   "(2) 00000 00000 00000 00000 00000 00000 00000 00000 00000 00100",
   "(2) 02000 00000 00000 00000 00000 00000 00000 00000 00000 00000",
   "(2) 00000 00000 00011 00200 00000 00000 00100 00000 00000 00000",
   "(2) 00000 00000 00022 00200 02000 00000 00200 00000 00000 00000",
   "(2) 01000 00000 00000 00000 00000 00000 00000 00000 00000 00000",
   "(2) 02000 00000 00000 00000 00000 00000 00000 00000 00000 00000",
   "(2) 02000 00000 00000 00000 00000 00000 00000 00000 00000 00000",
   "(2) 02000 00000 00010 00002 00000 00000 20020 00000 00000 00000",
   "(2) 02220 20222 22022 20202 02222 22200 20220 00002 00000 00000",
   "(2) 02220 20110 00022 20202 02222 22200 20220 00002 00000 00000",
   "(2) 02000 00000 00000 00000 00000 00000 00000 00000 00000 00000");
DECLARE NC1TRIPLES LITERALLY '262';
DECLARE C1TRIPLES(NC1TRIPLES) FIXED INITIAL ( 76847, 77359, 77574, 80131,
   199687, 199688, 199692, 199696, 199723, 199727, 208646, 265223, 265224,
   265228, 265232, 265259, 265263, 274182, 276739, 461831, 461832, 461836,
   461840, 461867, 461871, 470790, 527367, 527368, 527372, 527376, 527403,
   527407, 536326, 592903, 592904, 592908, 592912, 592939, 592943, 601862,
   658439, 658440, 658444, 658448, 658475, 658479, 667398, 723975, 723976,
   723980, 723984, 724011, 724015, 732934, 789511, 789512, 789516, 789520,
   789547, 789551, 798470, 801027, 803853, 803862, 803863, 803864, 803865,
   803866, 803867, 920583, 920584, 920588, 920592, 920619, 920623, 929542,
   934934, 934935, 934936, 934937, 934938, 934939, 1051655, 1051656, 1051660,
   1051664, 1051691, 1051695, 1060614, 1066006, 1066007, 1066008, 1066009,
   1066010, 1066011, 1126150, 1128707, 1313799, 1313800, 1313804, 1313808,
   1313835, 1313839, 1322758, 1328150, 1328151, 1328152, 1328153, 1328154,
   1328155, 1379335, 1379336, 1379340, 1379344, 1379371, 1379375, 1388294,
   1393686, 1393687, 1393688, 1393689, 1393690, 1393691, 1847046, 1849603,
   1903623, 1903624, 1903628, 1903632, 1903659, 1903663, 1912582, 2034695,
   2034696, 2034700, 2034704, 2034731, 2034735, 2043654, 2049039, 2558983,
   2558984, 2558988, 2558992, 2559019, 2559023, 2567942, 3345415, 3345416,
   3345420, 3345424, 3345451, 3345455, 3354374, 3484719, 3485231, 3542023,
   3542024, 3542028, 3542032, 3542059, 3542063, 3550982, 3556374, 3556375,
   3556376, 3556377, 3556378, 3556379, 3804167, 3804168, 3804172, 3804176,
   3804203, 3804207, 3813126, 4066311, 4066312, 4066316, 4066320, 4066347,
   4066351, 4075270, 4077827, 4080670, 4140806, 4143362, 4143373, 4206342,
   4208899, 4260112, 4260124, 4260127, 4260130, 4260131, 4260134, 4260136,
   4260138, 4260143, 4260144, 4260145, 4337410, 4599554, 4787207, 4787208,
   4787212, 4787216, 4787243, 4787247, 4796166, 4801538, 4801549, 5114887,
   5114888, 5114892, 5114896, 5114923, 5114927, 5123846, 5129218, 5129229,
   5573639, 5573640, 5573644, 5573648, 5573675, 5573679, 5582598, 5648130,
   5648141, 5713670, 5716227, 5901319, 5901320, 5901324, 5901328, 5901355,
   5901359, 5910278, 5915650, 5915653, 6106882, 6106893, 6622215, 6622216,
   6622220, 6622224, 6622251, 6622255, 6631174, 7220994, 7286530, 7536912,
   7536924, 7536927, 7536930, 7536931, 7536934, 7536936, 7536938, 7536943,
   7536944, 7536945);
DECLARE PRTB(132) FIXED INITIAL (0, 13663, 16943, 17967, 22063, 23855, 28207,
   28463, 16185, 25927, 18756, 18733, 18782, 20036, 23108, 25927, 23108, 47,
   0, 11311, 11823, 34, 40, 42, 47, 22063, 23855, 16185, 18756, 18733, 18782,
```

```
    3185, 3190, 3140, 20036, 8004, 16641, 65, 0, 0, 0, 0, 0, 0, 15933, 15940,
    4209, 0, 0, 0, 0, 0, 0, 41, 41, 66, 70, 44, 46, 110, 111, 0, 0, 0, 30727,
    30728, 8, 7, 0, 82, 0, 0, 0, 12809, 12810, 0, 0, 0, 64, 16641, 29441, 87,
    62, 0, 197689, 57, 19203, 772, 17466, 51, 39, 29, 85, 0, 24321, 0, 0, 0,
    28942, 0, 4745043, 80, 0, 0, 0, 0, 0, 0, 0, 14347, 0, 20245, 20, 0, 69, 0,
    17, 0, 0, 0, 0, 0, 0, 0, 54, 0, 0, 0, 28, 0, 0, 0, 0);
DECLARE PRDTB(132) BIT(8) INITIAL (0, 3, 29, 76, 92, 96, 101, 104, 123, 79,
    129, 130, 131, 58, 37, 77, 35, 36, 78, 91, 95, 122, 127, 128, 57, 90, 94,
    121, 124, 125, 126, 65, 54, 55, 56, 109, 9, 8, 67, 68, 69, 70, 71, 72,
    113, 40, 114, 118, 5, 28, 75, 107, 52, 100, 103, 27, 74, 89, 93, 99, 102,
    33, 21, 22, 42, 43, 44, 45, 41, 108, 1, 15, 4, 47, 48, 46, 51, 110, 2, 11,
    82, 111, 112, 10, 32, 30, 80, 31, 66, 117, 119, 132, 39, 81, 7, 6, 88, 12,
    60, 59, 115, 105, 16, 18, 20, 53, 87, 17, 19, 50, 49, 63, 62, 61, 73, 86,
    120, 13, 83, 97, 24, 84, 98, 23, 106, 64, 85, 26, 116, 14, 34, 38, 25);
DECLARE HDTB(132) BIT(8) INITIAL (0, 64, 66, 70, 86, 93, 110, 111, 63, 101,
    73, 73, 73, 78, 90, 100, 119, 90, 101, 86, 93, 63, 73, 73, 78, 102, 76,
    88, 97, 97, 97, 99, 98, 98, 91, 82, 60, 60, 58, 58, 58, 58, 58, 87,
    85, 62, 51, 53, 66, 70, 54, 98, 110, 111, 112, 69, 116, 92, 106, 109, 57,
    104, 104, 120, 120, 120, 120, 120, 89, 55, 104, 64, 50, 50, 50, 98, 77,
    52, 65, 121, 77, 59, 65, 67, 118, 71, 67, 114, 103, 83, 94, 68, 75, 95,
    95, 108, 61, 113, 113, 84, 96, 104, 104, 104, 98, 108, 104, 104, 56, 56,
    79, 79, 79, 107, 105, 81, 61, 115, 117, 74, 115, 117, 74, 80, 99, 105, 74,
    72, 104, 57, 68, 74);
DECLARE PRLENGTH(132) BIT(8) INITIAL (0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    3, 3, 3, 2, 1, 3, 3, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2,
    1, 1, 1, 1, 1, 1, 3, 3, 3, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1,
    1, 3, 3, 2, 2, 1, 2, 1, 1, 1, 3, 3, 1, 1, 1, 2, 3, 3, 2, 2, 1, 4, 2, 3, 3,
    3, 2, 2, 2, 2, 1, 3, 1, 1, 1, 3, 1, 4, 2, 1, 1, 1, 1, 1, 1, 1, 3, 1, 3, 2,
    1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 1, 1, 1);
DECLARE CONTEXT_CASE(132) BIT(8) INITIAL (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
DECLARE LEFT_CONTEXT(0) BIT(8) INITIAL ( 0);
DECLARE LEFT_INDEX(72) BIT(8) INITIAL ( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1);
DECLARE CONTEXT_TRIPLE(0) FIXED INITIAL ( 0);
DECLARE TRIPLE_INDEX(72) BIT(8) INITIAL ( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1);
DECLARE PR_INDEX(121) BIT(8) INITIAL ( 1, 2, 15, 15, 15, 17, 19, 19, 19, 19,
    19, 19, 25, 35, 35, 36, 36, 36, 36, 38, 38, 38, 39, 40, 41, 42, 43, 44,
    44, 44, 46, 46, 47, 48, 48, 49, 50, 51, 52, 52, 52, 52, 52, 53, 54, 54,
```

55, 62, 63, 64, 69, 69, 72, 73, 73, 73, 76, 77, 77, 78, 79, 84, 84, 84,
84, 84, 84, 86, 94, 94, 94, 94, 94, 94, 96, 96, 97, 98, 98, 100, 100, 102,
102, 102, 103, 103, 103, 103, 104, 105, 105, 106, 107, 107, 107, 107, 108,
109, 111, 114, 115, 115, 116, 116, 118, 119, 120, 121, 122, 123, 123, 123,
124, 125, 126, 126, 127, 128, 130, 131, 132, 133);

```
FIRST_OR_2ND_MATCH:PROCEDURE(A,B) BIT(1);
/* THIS PROCEDURE REQUIRES TWO CHARACTER PARAMETERS
   PARAMETER A IS THE UNDECLARED IDENTIFIER, PARAMETER B IS THE
   COMPARISION WORD -- IT RETURNS TRUE(1)/FALSE(0) */
 DECLARE (A,B) CHARACTER;
  IF (SUBSTR(A,0,1)=SUBSTR(B,0,1))|(SUBSTR(A,0,1)=SUBSTR(B,1,1))|
     (SUBSTR(A,1,1)=SUBSTR(B,0,1))|(SUBSTR(A,1,1)=SUBSTR(B,1,1))
  THEN RETURN TRUE;ELSE RETURN FALSE;
  END FIRST_OR_2ND_MATCH;
TRANSPOSED:PROCEDURE(A,B) BIT(1);
/* THIS PROCEDURE REQUIRES TWO CHARACTER PARAMETERS
   PARAMETER A IS THE UNDECLARED IDENTIFIER, PARAMETER B IS THE
   COMPARISION WORD -- IT RETURNS TRUE(1)/FALSE(0) */
 DECLARE (A,B) CHARACTER,(I,J) FIXED;
  DECLARE K BIT(1);
  DO; I=LENGTH(A); J=0;
  K=TRUE;
  DO WHILE K&(SUBSTR(A,J,1)=SUBSTR(B,J,1));
  J=J+1;
  IF J >= I THEN K=FALSE;
  END; I=J+1;
  IF (SUBSTR(A,J,1)=SUBSTR(B,I,1)) THEN
  IF (SUBSTR(A,I,1)=SUBSTR(B,J,1)) THEN RETURN TRUE; ELSE RETURN FALSE; END;
 END TRANSPOSED;
NUMBER_OF_ZEROS:PROCEDURE(A,B);
/* THIS PROCEDURE REQUIRES TWO CHARACTER PARAMETERS
   PARAMETER A IS THE UNDECLARED IDENTIFIER, PARAMETER B IS THE
   COMPARISION WORD -- IT RETURNS TRUE(1)/FALSE(0) */
 DECLARE (A,B) CHARACTER,(I,J,K) FIXED;
  DO; I=LENGTH(A); J=LENGTH(B);
  IF I<J THEN I=I; ELSE I=J;
  J=0;I=I-1;
  DO K=0 TO I;
  IF SUBSTR(A,K,1)¬=SUBSTR(B,K,1) THEN J=J+1; END;
  RETURN J;
  END;
  END NUMBER_OF_ZEROS;
LENGTHOK:PROCEDURE(A,B) BIT(1);
/* THIS PROCEDURE REQUIRES TWO CHARACTER PARAMETERS
   PARAMETER A IS THE UNDECLARED IDENTIFIER, PARAMETER B IS THE
   COMPARISION WORD -- IT RETURNS TRUE(1)/FALSE(0) */
 DECLARE (A,B) CHARACTER,(I,J,K,L) FIXED;
  DO; I=LENGTH(A); J=LENGTH(B);
  K=I-1; L=I+1;
/* AN IDENTIFIER IS CONSIDERED ACCEPTABLE IN LENGTH IF IT IS EQUAL TO OR
   DIFFERS BY ONLY ONE FROM THAT OF THE UNDECLARED IDENTIFIER */
 IF (I=J)|(J=K)|(J=L) THEN RETURN TRUE; ELSE RETURN FALSE; END;
  END LENGTHOK;
```

```
 SHIFT_1_TEST:PROCEDURE(A,B) BIT(1);
 /* THIS PROCEDURE REQUIRES TWO CHARACTER PARAMETERS
    PARAMETER A IS THE UNDECLARED IDENTIFIER, PARAMETER B IS THE
    COMPARISION WORD -- IT RETURNS TRUE(1)/FALSE(0) */
  DECLARE (A,B) CHARACTER,(I,J,K,L) FIXED;
   DECLARE M BIT(1);
   DO; I=LENGTH(A); J=LENGTH(B);
   IF I<J THEN L=I;ELSE L=J; K=0;
   M=TRUE;
   DO WHILE M&(SUBSTR(A,K,1)=SUBSTR(B,K,1));
   K=K+1; IF K >= L THEN M=FALSE; END;
   IF L=K THEN RETURN TRUE;
   IF I < J THEN DO; L=L-K; I=K+1;
   IF SUBSTR(A,K,L)=SUBSTR(B,I,L) THEN RETURN TRUE; ELSE RETURN FALSE;
   END; ELSE DO; L=L-K; I=K+1;
  IF SUBSTR(A,I,L)=SUBSTR(B,K,L) THEN RETURN TRUE; ELSE RETURN FALSE;
   END; END;
  END SHIFT_1_TEST;
 MISSPELLED_IDENTIFIER:PROCEDURE(A);
 /* THIS PROCEDURE REQUIRES 1 PARAMETER, A CHARACTER OF LENGTH >=2
    THE PROCEDURE SEARCHES THE SYMBOL TABLE AND RESERVE WORD LIST TO ATTEMPT
    TO MATCH THE UNDECLARED IDENTIFIER PASSED TO IT WITH ONE OF THE KNOWN
    IDENTIFIERS OR RESERVE WORDS -- THE PROCEDURE RETURNS THE SYMBOL TABLE
    ADDRESS IF FOUND, 0 IF IT STILL CAN'T IDENTIFY THE IDENTIFIER AND
    257+I WHERE I WILL IDENTIFY THE RESERVE WORD FOUND */
  DECLARE A CHARACTER,(I,J) FIXED;
   I=SY; IF LENGTH(A)=1 THEN RETURN 0;
   IF DECL_TEST THEN RETURN 0;
IF RESERVE THEN DO; RESERVE=FALSE; GO TO RESERVE_WORD; END;
   DO WHILE I>0;
   IF (FIRST_OR_2ND_MATCH(A,SYMBOL(I)))&(LENGTHOK(A,SYMBOL(I))) THEN
   IF (LENGTH(A)=LENGTH(SYMBOL(I))) THEN DO;
   J=NUMBER_OF_ZEROS(A,SYMBOL(I));
   IF J=1 THEN RETURN I;
   IF(J=2)&(TRANSPOSED(A,SYMBOL(I))) THEN RETURN I;
   END;
   ELSE IF SHIFT_1_TEST(A,SYMBOL(I)) THEN RETURN I;
   I=I-1;
   END;
 /* RESERVED WORD? */
RESERVE_WORD:
   IF LENGTH(A)>11 THEN RETURN  0;
   DO J=V_INDEX(LENGTH(A)-2) TO V_INDEX(LENGTH(A)+1)-1;
   IF (FIRST_OR_2ND_MATCH(A,V(J)))&(LENGTHOK(A,V(J))) THEN
   IF(LENGTH(A)=LENGTH(V(J))) THEN DO;
   I=NUMBER_OF_ZEROS(A,V(J));
   IF I=1 THEN RETURN 243+J;
   IF (I=2)&(TRANSPOSED(A,V(J))) THEN RETURN 243+J;
```

```
    END;
    ELSE IF SHIFT_1_TEST(A,V(J)) THEN RETURN 243+J;
    END;
    RETURN 0;
   END MISSPELLED_IDENTIFIER;
    LAFP_PROC:PROCEDURE(A);
    DECLARE (A,B) CHARACTER;
B='END';
     IF LENGTHOK(A,B) THEN DO;
     IF NUMBER_OF_ZEROS(A,B)=1 THEN DO; BCD=B; TOKEN=19; GO TO BTMM; END;
    ELSE IF NUMBER_OF_ZEROS(A,B)=2 & TRANSPOSED(A,B) THEN DO;
    BCD=B; TOKEN=19; GO TO BTMM; END; ELSE DO;
     IF SHIFT_1_TEST(A,B) THEN DO; BCD=B; TOKEN=19; GO TO BTMM; END;
    END;   END;
B='BEGIN';
     IF LENGTHOK(A,B) THEN DO;
     IF NUMBER_OF_ZEROS(A,B)=1 THEN DO; BCD=B; TOKEN=37; GO TO BTMM; END;
    ELSE IF NUMBER_OF_ZEROS(A,B)=2 & TRANSPOSED(A,B) THEN DO;
    BCD=B; TOKEN=37; GO TO BTMM; END; ELSE DO;
     IF SHIFT_1_TEST(A,B) THEN DO; BCD=B; TOKEN=37; GO TO BTMM; END;
    END;   END;
    B='LOCAL';
     IF LENGTHOK(A,B) THEN DO;
     IF NUMBER_OF_ZEROS(A,B)=1 THEN DO; BCD=B; TOKEN=38; GO TO BTMM; END;
    ELSE IF NUMBER_OF_ZEROS(A,B)=2 & TRANSPOSED(A,B) THEN DO;
    BCD=B; TOKEN=38; GO TO BTMM; END; ELSE DO;
     IF SHIFT_1_TEST(A,B) THEN DO; BCD=B; TOKEN=38; GO TO BTMM; END;
    END;   END;
    B='ARRAY';
     IF LENGTHOK(A,B) THEN DO;
     IF NUMBER_OF_ZEROS(A,B)=1 THEN DO; BCD=B; TOKEN=39; GO TO BTMM; END;
    ELSE IF NUMBER_OF_ZEROS(A,B)=2 & TRANSPOSED(A,B) THEN DO;
    BCD=B; TOKEN=39; GO TO BTMM; END; ELSE DO;
     IF SHIFT_1_TEST(A,B) THEN DO; BCD=B; TOKEN=39; GO TO BTMM; END;
    END;   END;
    B='FUNCTION';
     IF LENGTHOK(A,B) THEN DO;
     IF NUMBER_OF_ZEROS(A,B)=1 THEN DO; BCD=B; TOKEN=49; GO TO BTMM; END;
    ELSE IF NUMBER_OF_ZEROS(A,B)=2 & TRANSPOSED(A,B) THEN DO;
    BCD=B; TOKEN=49; GO TO BTMM; END; ELSE DO;
     IF SHIFT_1_TEST(A,B) THEN DO; BCD=B; TOKEN=49; GO TO BTMM; END;
    END;   END;
     B='PROCEDURE';
     IF LENGTHOK(A,B) THEN DO;
     IF NUMBER_OF_ZEROS(A,B)=1 THEN DO; BCD=B; TOKEN=51; GO TO BTMM; END;
    ELSE IF NUMBER_OF_ZEROS(A,B)=2 & TRANSPOSED(A,B) THEN DO;
    BCD=B; TOKEN=51; GO TO BTMM; END; ELSE DO;
     IF SHIFT_1_TEST(A,B) THEN DO; BCD=B; TOKEN=51; GO TO BTMM; END;
```

```
    END;  END;
RETURN;
    BTMM:OUTPUT='**** ERROR ****  MISPELLED RESERVED WORD  '||A||'  REPLACED BY
     '||B;
    RETURN;
    END LAFP_PROC;
```

```
SCAN:
   PROCEDURE;
      DECLARE (S1, S2) FIXED;
      CALLCOUNT(3) = CALLCOUNT(3) + 1;
      FAILSOFT = TRUE;
      BCD = '';  NUMBER_VALUE = 0;
   SCAN1:
      DO FOREVER;
         IF CP > TEXT_LIMIT THEN CALL GET_CARD;
         ELSE
            DO; /* DISCARD LAST SCANNED VALUE */
               TEXT_LIMIT = TEXT_LIMIT - CP;
               TEXT = SUBSTR(TEXT, CP);
               CP = 0;
            END;
         /*  BRANCH ON NEXT CHARACTER IN TEXT                    */
         DO CASE CHARTYPE(BYTE(TEXT));

            /*  CASE 0  */

            /* ILLEGAL CHARACTERS FALL HERE  */
            CALL ERROR ('ILLEGAL CHARACTER: ' || SUBSTR(TEXT, 0, 1));

            /*  CASE 1  */

            /*  BLANK  */
            DO;
               CP = 1;
               DO WHILE BYTE(TEXT, CP) = BYTE(' ') & CP <= TEXT_LIMIT;
                  CP = CP + 1;
               END;
               CP = CP - 1;
            END;

            /*  CASE 2  */

            DO; /*  LOCATE THE STRING AND PLACE INTO BCD */
            TOKEN=STRING; BCD='';
                DO FOREVER;
                    DO CP=CP+1 TO TEXT_LIMIT;
                    S1=BYTE(TEXT,CP); IF S1=BYTE('"') THEN
                        DO; CP=CP+1; RETURN;
                        END;
                    BCD=BCD||SUBSTR(TEXT,CP,1); /* SLOW, BUT FIX LATER */
                    END;
                CALL GET_CARD; /* 'CAUSE WE WENT OFF THEN END OF THE CARD */
                END;
            END;
```

```
      /*  CASE 3  */
   ;  /*  NOT USED IN SKELETON (BUT USED IN XCOM)  */

      /*  CASE 4  */

      DO FOREVER;  /* A LETTER:  IDENTIFIERS AND RESERVED WORDS */
         DO CP = CP + 1 TO TEXT_LIMIT;
            IF NOT_LETTER_OR_DIGIT(BYTE(TEXT, CP)) THEN
               DO;  /* END OF IDENTIFIER  */
                  IF CP > 0 THEN BCD = BCD || SUBSTR(TEXT, 0, CP);
                  S1 = LENGTH(BCD);
IF INLINEV= 1 THEN DO;
                  IF S1=3 THEN /* POSSIBLY AN OP/DATA CODE */
                      DO;  I=SEARCH(STORECODE1,BCD);
                      IF I>=0 THEN
                          DO; TOKEN=DATADESC;
                          NUMBER_VALUE=I;
                          RETURN;
                          END;
                      I=SEARCH(OPCODE,BCD);
                      IF I>=0 THEN
                          DO; TOKEN=OPERATION;
                          NUMBER_VALUE=I;
                          RETURN;
                          END;
                      END;
  END;
                  IF S1 > 1 THEN IF S1 <= RESERVED_LIMIT THEN
                     /* CHECK FOR RESERVED WORDS */
                     DO I = V_INDEX(S1-1) TO V_INDEX(S1) - 1;
                        IF BCD = V(I) THEN
                           DO;
                              TOKEN = I;
                              RETURN;
                           END;
                     END;
                  /*  RESERVED WORDS EXIT HIGHER: THEREFORE <IDENTIFIER>*/
                  IF BCD='COMMENT' THEN
                      DO; S1=0;
                          DO WHILE S1¬=BYTE(';'); CALL CHAR;
                          S1=BYTE(TEXT,CP);
                          END;
                      BCD=''; NUMBER_VALUE=0; CP=CP+1; GO TO SCAN1;
                      END;
                  TOKEN = IDENT;
                  RETURN;
               END;
```

```
                END;
                /*  END OF CARD  */
                BCD = BCD || TEXT;
                CALL GET_CARD;
                CP = -1;
             END;

             /*  CASE 5  */

             DO;       /*  DIGIT:  A NUMBER  */
                TOKEN = NUMBER;
                DO FOREVER;
                   DO CP = CP TO TEXT_LIMIT;
                      S1 = BYTE(TEXT, CP);
                      IF S1 < "F0" THEN RETURN;
                      NUMBER_VALUE = 10*NUMBER_VALUE + S1 - "F0";
                   END;
                   CALL GET_CARD;
                END;
             END;

             /*  CASE 6  */
                 ;

             /*  CASE 7  */
             DO;       /*  SPECIAL CHARACTERS  */
                TOKEN = TX(BYTE(TEXT));
                CP = 1;
                RETURN;
             END;

             /*  CASE 8  */
          ;  /*  NOT USED IN SKELETON (BUT USED IN XCOM)  */
           END;      /* OF CASE ON CHARTYPE  */
           CP = CP + 1;  /* ADVANCE SCANNER AND RESUME SEARCH FOR TOKEN  */
        END;
    END SCAN;
    CLEAN_STACK:PROCEDURE;
    DO WHILE PARSE_STACK(SP) ¬= TX(BYTE(';'));
    IF SP <=2 THEN GO TO NEXT_CALL_SCAN;
SP=SP-1; END; SP=SP-1;
    NEXT_CALL_SCAN: DO WHILE TOKEN ¬= TX(BYTE(';'));
    CALL SCAN; END;
    IF ¬ CODE_STOPPED THEN
    OUTPUT='**** ERROR **** UNDECODABLE STATEMENT -- CODE GENERATION ABORTED';
   ELSE OUTPUT='**** ERROR **** UNDECODABLE STATEMENT -- STATEMENT DELETED';
CODE_STOPPED,CODE_GEN=FALSE;
   END CLEAN_STACK;
```

```
RIGHT_CONFLICT:
   PROCEDURE (LEFT) BIT(1);
      DECLARE LEFT FIXED;
      /*  THIS PROCEDURE IS TRUE IF TOKEN IS ¬ A LEGAL RIGHT CONTEXT OF LEFT*/
      RETURN ("CO" & SHL(BYTE(C1(LEFT), SHR(TOKEN,2)), SHL(TOKEN,1)
         & "06")) = 0;
   END RIGHT_CONFLICT;
RECOVER:
   PROCEDURE;
      IF ¬FAILSOFT THEN CALL SCAN; FAILSOFT=TRUE;
        DO WHILE FAILSOFT;
        IF SP=1 THEN
            DO; SP=SP+1; PARSE_STACK(SP)=BEGINV;
            SCANSEMI:
                DO WHILE TOKEN¬=SEMI; CALL SCAN;
                END;
            CALL SCAN; FAILSOFT=FALSE;
            END; ELSE
        IF PARSE_STACK(SP)=SEMI THEN GO TO SCANSEMI; ELSE
        IF PARSE_STACK(SP)=ENDV THEN
            DO; DO WHILE (TOKEN¬=SEMI)|(TOKEN¬=ENDV); CALL SCAN;
                END; FAILSOFT=FALSE;
            END; ELSE SP=SP-1;
        END;
      OUTPUT = 'RESUME:' || SUBSTR(POINTER, TEXT_LIMIT-CP+MARGIN_CHOP+7);
   END RECOVER;
STACKING:
   PROCEDURE BIT(1);  /* STACKING DECISION FUNCTION */
  DECLARE SPERROR CHARACTER,FIRST_FIX BIT(1);
      CALLCOUNT(1) = CALLCOUNT(1) + 1;
      DO FOREVER;     /* UNTIL RETURN  */
   FIRST_FIX=FALSE;
   TOPSTKNG:IF TOKEN=ELSEV THEN /*KLUDGE TO HANDLE ELSES */
                DO; IF PARSE_STACK(SP-1)=IFCLAUSE &
                PARSE_STACK(SP)=STATEMENT THEN RETURN TRUE;
                END;
IF TOKEN ¬=EOFILE THEN DO;
  /* SPECIAL TEST FOR DECLARATIONS  */
    IF LAFP_TEST & TOKEN=IDENT THEN CALL LAFP_PROC(BCD);
  /*  SPECIAL TEST FOR END  */
   IF PARSE_STACK(SP-1)=TX(BYTE(';')) THEN
   IF PARSE_STACK(SP)=IDENT THEN
   IF TOKEN=TX(BYTE(';')) THEN DO;  I5=MISSPELLED_IDENTIFIER(VAR(SP));
   SPERROR=VAR(SP);
   IF I5-257=5 THEN GO TO KLUDGE; END;
  /*  SPECIAL TEST FOR <IDENTIFIER>  (  */
    IF PARSE_STACK(SP)=IDENT THEN
IF TOKEN=TX(BYTE('(')) THEN DO; RESERVE=TRUE; I5=LOOKUP(VAR(SP));
```

```
SPERROR=VAR(SP);
   IF I5=0 THEN DO; I5=MISSPELLED_IDENTIFIER(VAR(SP)); IF I5>=257 THEN
   GO TO KLUDGE; END; END;
END;
         DO CASE SHR(BYTE(C1(PARSE_STACK(SP)),SHR(TOKEN,2)),SHL(3-TOKEN,1)&6)&3;
            /*  CASE 0  */
            DO;    /* ILLEGAL SYMBOL PAIR  */
  /*  IF 2ND ATTEMPT ERROR  */
   IF FIRST_FIX THEN GO TO ERRORCALL;
  /*  CHECK PARSE STACK THEN TOKEN FOR SPELLING ERROR  */
   IF (TOKEN=IDENT)|(PARSE_STACK(SP)=IDENT) THEN DO;
   I5=0; I5=MISSPELLED_IDENTIFIER(VAR(SP));
   SPERROR=VAR(SP);
   IF I5>=257 THEN GO TO KLUDGE;
   I5=0; I5=MISSPELLED_IDENTIFIER(BCD);
   SPERROR=BCD;
   IF I5>=257 THEN GO TO KLUDGE; ELSE GO TO ERRORCALL;
   KLUDGE:OUTPUT='**** ERROR **** MISSPELLED RESERVE WORD '||SPERROR||' REPLACED
   BY  '||V(I5-243);
  /*  SPECIAL TEST CASE STATEMENT FOR MISSPELLING CORRECTION  */
   DO CASE I5-257;
 /* CASE  0     --      OR    */
   DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  1     --      OF    */
   DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  2     --      IF    */
   DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  3     --      DO    */
   DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  4     --      _|_   */
   ;
 /* CASE  5     --      END   */
   DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  6     --      AND   */
   DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  7     --      NOT   */
   DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  8     --      LSS   */
   DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  9     --      LEQ   */
   DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  10    --      EQL   */
   DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  11    --      NEQ   */
   DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  12    --      GEQ   */
   DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
 /* CASE  13    --      GTR   */
```

```
  DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  14    --      FOR    */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  15    --      TAB    */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  16    --      FUN    */
  ;
/* CASE  17    --      ELSE   */
  DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  18    --      CASE   */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  19    --      THEN   */
  DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  20    --      STEP   */
  DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  21    --      READ   */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  22    --      PROC   */
  ;
/* CASE  23    --      BEGIN  */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  24    --      LOCAL  */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  25    --      ARRAY  */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  26    --      WHILE  */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  27    --      UNTIL  */
  DO; TOKEN=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  28    --      WRITE  */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  29    --      DOONCE  */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  30    --      INLINE  */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  31    --      WRITEON  */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  32    --      DOWHILE  */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  33    --      FORWARD  */
  ;
/* CASE  34    --      <NUMBER>  */
  ;
/* CASE  35    --      FUNCTION  */
  DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
/* CASE  36    --      <STRING>  */
  ;
/* CASE  37    --      PROCEDURE  */
```

```
      DO; PARSE_STACK(SP)=I5-243; FIRST_FIX=TRUE; GO TO TOPSTKNG; END;
      END;     END;
      ELSE DO; ERRORCALL:
                  CALL ERROR('ILLEGAL SYMBOL PAIR: ' || V(PARSE_STACK(SP)) || X1 ||
                     V(TOKEN), 1);
                  CALL STACK_DUMP;
                  CALL RECOVER;
      END;
               END;
               /*  CASE 1  */
               RETURN TRUE;        /*  STACK TOKEN  */
               /*  CASE 2  */
               RETURN FALSE;       /* DON'T STACK IT YET  */
               /*  CASE 3  */
               DO;         /* MUST CHECK TRIPLES  */
                  J = SHL(PARSE_STACK(SP-1), 16) + SHL(PARSE_STACK(SP), 8) + TOKEN;
                  I = -1;  K = NC1TRIPLES + 1;  /* BINARY SEARCH OF TRIPLES  */
                  DO WHILE I + 1 < K;
                     L = SHR(I+K, 1);
                     IF C1TRIPLES(L) > J THEN K = L;
                     ELSE IF C1TRIPLES(L) < J THEN I = L;
                     ELSE RETURN TRUE;  /* IT IS A VALID TRIPLE  */
                  END;
                  RETURN FALSE;
               END;
            END;     /* OF DO CASE  */
         END;   /*  OF DO FOREVER */
      END STACKING;
PR_OK:
      PROCEDURE(PRD) BIT(1);
         /* DECISION PROCEDURE FOR CONTEXT CHECK OF EQUAL OR IMBEDDED RIGHT PARTS*/
         DECLARE (H, I, J, PRD) FIXED;
         DO CASE CONTEXT_CASE(PRD);

            /*  CASE 0 -- NO CHECK REQUIRED  */
            RETURN TRUE;
            /*  CASE 1 -- RIGHT CONTEXT CHECK  */
            RETURN ¬ RIGHT_CONFLICT (HDTB(PRD));
            /*  CASE 2 -- LEFT CONTEXT CHECK  */
            DO;
               H = HDTB(PRD) - NT;
               I = PARSE_STACK(SP - PRLENGTH(PRD));
               DO J = LEFT_INDEX(H-1) TO LEFT_INDEX(H) - 1;
                  IF LEFT_CONTEXT(J) = I THEN RETURN TRUE;
               END;
               RETURN FALSE;
            END;
            /*  CASE 3 -- CHECK TRIPLES  */
```

```
            DO;
               H = HDTB(PRD) - NT;
               I = SHL(PARSE_STACK(SP - PRLENGTH(PRD)), 8) + TOKEN;
               DO J = TRIPLE_INDEX(H-1) TO TRIPLE_INDEX(H) - 1;
                  IF CONTEXT_TRIPLE(J) = I THEN RETURN TRUE;
               END;
               RETURN FALSE;
            END;
         END;  /* OF DO CASE  */
      END PR_OK;
   REDUCE:
      PROCEDURE;
         DECLARE (I, J, PRD) FIXED;
   IF V(PARSE_STACK(2))='<BLOCK>' THEN DO;
      IF TOKEN=EOFILE THEN DO; I=SYNTHESIZE(1,64);
      PARSE_STACK(2)=64; RETURN;
      END; END;
         /* PACK STACK TOP INTO ONE WORD */
         DO I = SP - 4 TO SP - 1;
            J = SHL(J, 8) + PARSE_STACK(I);
         END;
         DO PRD = PR_INDEX(PARSE_STACK(SP)-1) TO PR_INDEX(PARSE_STACK(SP)) - 1;
            IF (PRMASK(PRLENGTH(PRD)) & J) = PRTB(PRD) THEN
               IF PR_OK(PRD) THEN
               DO;  /* AN ALLOWED REDUCTION */
                  MP = SP - PRLENGTH(PRD) + 1; MPP1 = MP + 1;
                  I=SYNTHESIZE(PRDTB(PRD),HDTB(PRD));
                  SP=MP; PARSE_STACK(SP)=I;
                  RETURN;
               END;
         END;

         /* LOOK UP HAS FAILED, ERROR CONDITION */
      IF OLD_LINE_NO ¬= CARD_COUNT & OLD_SP ¬= SP THEN DO;
      OLD_LINE_NO=CARD_COUNT; OLD_SP=SP;
         CALL ERROR('NO PRODUCTION IS APPLICABLE',1);
         CALL STACK_DUMP;
         FAILSOFT = FALSE;
         CALL RECOVER;
     END; ELSE CALL CLEAN_STACK;
      END REDUCE;
   COMPILATION_LOOP:
      PROCEDURE;
         COMPILING = TRUE;
         DO WHILE COMPILING;      /* ONCE AROUND FOR EACH PRODUCTION (REDUCTION)  */
            DO WHILE STACKING;
               SP = SP + 1;
               IF SP = STACKSIZE THEN
```

```
            DO;
               CALL ERROR ('STACK OVERFLOW *** CHECKING ABORTED ***', 2);
               RETURN;    /* THUS ABORTING CHECKING */
            END;
         PARSE_STACK(SP) = TOKEN;
         VAR(SP) = BCD;
         FIXV(SP) = NUMBER_VALUE;
         CALL SCAN;
      END;
      CALL REDUCE;
   END;      /* OF DO WHILE COMPILING  */
END COMPILATION_LOOP;
```

```
%ALGOL   10:00,20000
BEGIN INTEGER NS,CP; STRING(80) CARD;
INTEGER ARRAY PS1(1::3,0::300);
INTEGER ARRAY PS2(1::3,1::300); INTEGER TOT;
INTEGER MAX,FST,SND,TRD;
INTEGER TEMP,TENP;
INTEGER NP;
INTEGER CALL;
INTEGER MDSYM,RTSYM;
INTEGER POINTER;
INTEGER POINT;
INTEGER NT; NT:=49;
NP:=132;
COMMENT PROGRAM TO PRODUCE GRAMMAR NUMERICAL TABLES;
READ (NS);
BEGIN INTEGER ARRAY PRODUCTION(0::4,1::161);
          STRING(29) ARRAY VOCABULARY(0::NS);
LOGICAL ARRAY ALRDY(NT+1::NS); INTEGER CURSYMBOL,CURSUB;
BITS ARRAY LPS(1::2*NT); INTEGER TRKNT;
LOGICAL ARRAY ALRDX(   1::NS);
BITS ARRAY RIGHTP(NT+1::NS,1::2*NT);
INTEGER ARRAY FARRIGHT(1::NS+1);
INTEGER ARRAY FRT(1::200);
INTEGER ARRAY ONRIGHT(1::NS+1); INTEGER ARRAY RMOST(1::200);
INTEGER ARRAY DONE(NT+1::NS);
INTEGER ARRAY CURRENT(NT+1::NS);
BITS ARRAY RIGHTS(NT+1::NS,1::2);
INTEGER ARRAY SINGLE(NT+1::NS,1::6);
INTEGER ARRAY TER(1::500);  INTEGER ARRAY TERINDEX(1::NS+1);
INTEGER ARRAY NTERINDEX(NT+1::NS+1); INTEGER ARRAY NTER(1::200);
PROCEDURE SHIFT;
   FOR K:=1 UNTIL NP DO BEGIN INTEGER I; I:=4;
   WHILE PRODUCTION(I,K)=0 DO BEGIN
PRODUCTION(I  ,K):=PRODUCTION(I-1,K);
PRODUCTION(I-1,K):=PRODUCTION(I-2,K);
PRODUCTION(I-2,K):=PRODUCTION(I-3,K);
PRODUCTION(I-3,K):=0; END;
END;
PROCEDURE PSLEGAL(INTEGER VALUE A,B,C);
BEGIN
   FOR I:=1 UNTIL MAX DO
     IF (PS1(1,I)=A) AND (PS1(2,I)=B) AND (PS1(3,I)=C) THEN BEGIN
        PS2(1,TOT):=A; PS2(2,TOT):=B; PS2(3,TOT):=C; TOT:=TOT+1;
       PS1(1,I):=PS1(2,I):=PS1(3,I):=0;
       GO TO BOTPS; END;
BOTPS:END PSLEGAL;
PROCEDURE LOAD(INTEGER VALUE I);
```

```
BEGIN INTEGER K; K:=0;
   WHILE TRD=I DO
     BEGIN K:=K+1; PS1(1,K):=FST;
                   PS1(2,K):=SND;
                   PS1(3,K):=TRD;
     READON(FST,SND,TRD);
   END;
MAX:=K; FOR I:=K+1 UNTIL 300 DO PS1(1,I):=PS1(2,I):=PS1(3,I):=0;
FOR I:=1 UNTIL 300 DO PS2(1,I):=PS2(2,I):=PS2(3,I):=0;
TOT:=1;
END LOAD;
PROCEDURE SORT;
   BEGIN INTEGER A,B,C,D; LOGICAL E;
     RIPPLE:E:=FALSE; D:=TOT-2;
       FOR I:=1 UNTIL D DO
         IF PS2(1,I)>PS2(1,I+1) THEN
           BEGIN A:=PS2(1,I); B:=PS2(2,I); C:=PS2(3,I);
               PS2(1,I):=PS2(1,I+1);
               PS2(2,I):=PS2(2,I+1);
               PS2(3,I):=PS2(3,I+1);
               PS2(1,I+1):=A; PS2(2,I+1):=B; PS2(3,I+1):=C; E:=TRUE;
           END;
     IF E THEN IF D-1 ¬= 1 THEN
       BEGIN D:=D-1; GO TO RIPPLE; END;
     FOR I:=1 UNTIL TOT-1 DO
       WRITE(VOCABULARY(PS2(1,I)),VOCABULARY(PS2(2,I)),
             VOCABULARY(PS2(3,I)));
 END SORT;
PROCEDURE NEED2(INTEGER VALUE N);
IF (N<=NT) OR (¬ALRDX(N)) THEN
BEGIN
   FOR I:=TERINDEX(N) UNTIL TERINDEX(N+1)-1 DO BEGIN INTEGER Q,P;
ALRDX(N):=TRUE;
FOR LL:=NT+1 UNTIL NS DO ALRDY(LL):=FALSE;
Q:=TER(I);
   FOR R:=0 UNTIL 3 DO IF PRODUCTION(R+1,Q)=N THEN
   BEGIN
     IF R=0 THEN NEED2(PRODUCTION(0,Q))
              ELSE IF R=1 THEN BEGIN
                     IF PRODUCTION(R,Q)<=NT THEN BEGIN
                                                MDSYM:=PRODUCTION(R,Q);
                                                NEED1(PRODUCTION(0,Q)); END
                                ELSE BEGIN INTEGER L,M;
                                M:=PRODUCTION(1,Q);
                                FOR K:=1 UNTIL NT DO BEGIN L:=2*(K-1);
                                FOR X:=0 UNTIL 31 DO
                                IF (RIGHTP(M,L+1) SHR X) AND#1=#1 THEN
           PSLEGAL(K,X+1,RTSYM);
```

```
                                FOR X:=0 UNTIL 23 DO
                                IF (RIGHTP(M,L+2) SHR X) AND#1=#1 THEN
            PSLEGAL(K,X+33,RTSYM); END;
                                IF SINGLE(M,1)¬=0 THEN
                                FOR X:=1 UNTIL 6 DO
                                IF SINGLE(M,X)¬=0 THEN BEGIN
                                                    MDSYM:=SINGLE(M,X);
                                NEED1(PRODUCTION(0,Q)); END;
                                END;
               END
               ELSE BEGIN INTEGER L,M;M:=PRODUCTION(R,Q);
                    L:=PRODUCTION(R-1,Q);
IF M<=NT THEN BEGIN IF L<=NT THEN PSLEGAL(L,M,RTSYM)
                                ELSE BEGIN
                                FOR X:=0 UNTIL 31 DO
                                IF(RIGHTS(L,1) SHR X) AND #1=#1 THEN
            PSLEGAL(X+1,M,RTSYM);
                                FOR X:=0 UNTIL 23 DO
                                IF(RIGHTS(L,2) SHR X) AND #1=#1 THEN
            PSLEGAL(X+33,M,RTSYM);  END; END
                    ELSE BEGIN
                                FOR K:=1 UNTIL NT DO BEGIN L:=2*(K-1);
                                FOR X:=0 UNTIL 31 DO
                                IF (RIGHTP(M,L+1) SHR X) AND#1=#1 THEN
            PSLEGAL(K,X+1,RTSYM);
                                FOR X:=0 UNTIL 23 DO
                                IF (RIGHTP(M,L+2) SHR X) AND#1=#1 THEN
            PSLEGAL(K,X+33,RTSYM);  END;
IF SINGLE(M,1)¬=0 THEN BEGIN IF L<=NT THEN BEGIN
                                FOR X:=1 UNTIL 6 DO
                                IF SINGLE(M,X)¬=0 THEN
            PSLEGAL(L,SINGLE(M,X),RTSYM);
END
                    ELSE BEGIN
                                FOR Y:=1 UNTIL 6 DO
                                IF SINGLE(M,Y)¬=0 THEN BEGIN
                                FOR X:=0 UNTIL 31 DO
                                IF(RIGHTS(L,1) SHR X) AND. #1=#1 THEN
            PSLEGAL(X+1,SINGLE(M,Y),RTSYM);
                                FOR X:=0 UNTIL 31 DO
                                IF(RIGHTS(L,2) SHR X) AND #1=#1 THEN
            PSLEGAL(X+33,SINGLE(M,Y),RTSYM);   END;
                    END;
               END;
          END;
    END;
END;
END;
```

```
ALRDX(N):=FALSE; END;
PROCEDURE NEED1(INTEGER VALUE M);
IF ¬ALRDY(M) THEN
BEGIN
   FOR I:=TERINDEX(M) UNTIL TERINDEX(M+1)-1 DO BEGIN INTEGER Q;
ALRDY(M):=TRUE;
Q:=TER(I);
   FOR R:=0 UNTIL 3 DO IF PRODUCTION(R+1,Q)=M THEN BEGIN
         IF R=0 THEN NEED1(PRODUCTION(0,Q))
            ELSE BEGIN INTEGER N;
                  N:=PRODUCTION(R,Q);
                  IF N<=NT THEN PSLEGAL(N,MDSYM,RTSYM)
                  ELSE BEGIN
                              FOR X:=0 UNTIL 31 DO
                              IF (RIGHTS(N,1) SHR X) AND #1=#1 THEN
           PSLEGAL(X+1,MDSYM,RTSYM);
                              FOR X:=0 UNTIL 31 DO
                              IF (RIGHTS(N,2) SHR X) AND #1=#1 THEN
           PSLEGAL(X+33,MDSYM,RTSYM);          END;
               END;
      END;
END;
ALRDY(M):=FALSE; END;
PROCEDURE UNSHIFT;
FOR K:=1 UNTIL NP   DO BEGIN INTEGER I; I:=1;
     WHILE PRODUCTION(I,K)=0 DO BEGIN
       PRODUCTION(I  ,K):=PRODUCTION(I+1,K);
       PRODUCTION(I+1,K):=PRODUCTION(I+2,K);
       PRODUCTION(I+2,K):=PRODUCTION(I+3,K);
       PRODUCTION(I+3,K):=0; END;
   END;
          PROCEDURE SCAN; BEGIN VOCABULARY(0):=" ";
          COMMENT READ VOCABULARY INTO ARRAY AND
          LENGTHS INTO  LENGTH ARRAY --
          INITIALIZE PRODUCTIONS ARRAY TO 0;
FOR I:=0 UNTIL 4 DO FOR J:=1 UNTIL 160 DO PRODUCTION(I,J):=0;
PRODUCTION(0,161):=10000;
FOR I:=1 UNTIL NS DO BEGIN READCARD(CARD);VOCABULARY(I):=CARD(0|29);END;
          COMMENT  START DOWN THROUGH PRODUCTIONS;
            FOR K:=1 UNTIL 160 DO
BEGIN INTEGER L; STRING(29) T;
READCARD(CARD); TEMP:=0; L:=0;
WRITE(CARD);
T:="                                  ";
CP:=0;
               IF CARD(0|7) = "STOP IT" THEN GO TO STOP;
               IF CARD(0|1)="<" THEN
                 BEGIN INTEGER I;
```

```
                    CP:=0; I:=0;
                    WHILE CARD(I|1)¬=">" DO
                     BEGIN T(I|1):=CARD(CP|1);
                     CP:=CP+1; I:=I+1; END;
                    T(I|1):=">"; CP:=CP+1;
                    FOR I:=1 UNTIL NS DO
                      IF T(0|29)=VOCABULARY(I) (0|29) THEN
BEGIN TENP:=I; GO TO NEXT; END;
                    END;
NEXT:PRODUCTION(0,K):=TENP;
              TOP: L:=L+1;
T:="                              ";
                    WHILE (CP<80) AND (CARD(CP|1)=" ") DO CP:=CP+1;
IF CP = 80 THEN GO TO LOOP;
IF (CARD(CP|1)="<") AND (CARD(CP+1|1)¬=" ") THEN BEGIN INTEGER I;
                      I:=0;
                       WHILE (I<29) AND (CARD(CP|1)¬=">") DO
                         BEGIN T(I|1):=CARD(CP|1); I:=I+1;
                         CP:=CP+1; END; T(I|1):=">";
CP:=CP+1;
                       END
                    ELSE BEGIN INTEGER I; I:=0;
                         WHILE (I<29) AND (CARD(CP|1)¬=" ") DO
                           BEGIN T(I|1):=CARD(CP|1); I:=I+1;
                           CP:=CP+1; END; END;
FOR MI:=1 UNTIL NS DO
                    IF VOCABULARY(MI)  = T THEN BEGIN
TEMP:=MI; GO TO ADD; END;
ADD:PRODUCTION(L,K):=TEMP; GO TO TOP;
LOOP:
WRITE(" "); FOR I:=0 UNTIL 4 DO IF PRODUCTION(I,K)
                    ¬=0 THEN WRITEON (PRODUCTION(I,K));
            BOTTOM:END;
         END SCAN;
         SCAN;
STOP:BEGIN
PROCEDURE FALNAS(INTEGER VALUE J);
    FOR I:=FARRIGHT(J) UNTIL FARRIGHT(J+1)-1 DO
      BEGIN INTEGER C,R; R:=PRODUCTION(3,FRT(I));
        IF R=0 THEN
          FALNAS(PRODUCTION(0,FRT(I)))
         ELSE IF R<=NT  THEN
          IASBRPS(R,PRODUCTION(0,FRT(I)))
                          ELSE
BEGIN
FOR II:=0 UNTIL 31 DO IF (RIGHTS(R,1) SHR II) AND #1 =#1 THEN
   IASBRPS(II+1,PRODUCTION(0,FRT(I)));
FOR II:=0 UNTIL 31 DO IF (RIGHTS(R,2) SHR II) AND #1 =#1 THEN
```

```
   IASBRPS(II+33,PRODUCTION(0,FRT(I)));
   END;   END;
PROCEDURE IASBRPS(INTEGER VALUE J,X);
   BEGIN INTEGER Q,R;
PROCEDURE BRPS( INTEGER VALUE N);
   BEGIN
     IF ALRDY(N) THEN GO TO RTN
        ELSE BEGIN ALRDY(N):=TRUE;
RIGHTP(N,Q):=RIGHTP(N,Q) OR (#1 SHL R);
                END;
     FOR JJ:=ONRIGHT(N) UNTIL ONRIGHT(N+1)-1 DO BRPS(RMOST(JJ));
RTN: END BRPS;
IF (CURSYMBOL <= 32) THEN R:=CURSYMBOL-1 ELSE R:=CURSYMBOL-33;
IF CURSYMBOL <= 32 THEN Q:=2*(J-1)+1 ELSE Q:=2*(J-1)+2;
   FOR G:=NT+1 UNTIL NS DO ALRDY(G):=FALSE;
      BRPS(X);
 END IASBRPS;
      PROCEDURE BRS(INTEGER VALUE N);
         BEGIN
            IF N>NT THEN
               IF ALRDY(N) THEN GO TO RETN
                     ELSE BEGIN ALRDY(N) := TRUE;
                           RIGHTS(N,CURSUB) := RIGHTS(N,CURSUB) OR
                              (#1 SHL ((CURSYMBOL-1) REM 32));
                     END;
           FOR J:= ONRIGHT(N) UNTIL ONRIGHT(N+1)-1 DO  BRS(RMOST(J));
         RETN:  END BRS;
PROCEDURE OTHER(INTEGER VALUE I,J);
BEGIN
   FOR K:=TERINDEX(I) UNTIL TERINDEX(I+1)-1 DO BEGIN INTEGER L;
       L:=TER(K); IF (PRODUCTION(1,L)=I) AND (PRODUCTION(2,L)=0) THEN
         BEGIN INTEGER Q,R; Q:=1; R:=PRODUCTION(0,L);
         WHILE (Q<=6) AND(SINGLE(R,Q)¬=J) AND (SINGLE(R,Q)¬=0) DO
         Q:=Q+1; IF (Q=7) THEN WRITE("OOPS") ELSE IF SINGLE(R,Q)¬=J THEN
         SINGLE(R,Q):=J;
     OTHER(R,J);
   END;  END; END;
PROCEDURE SGL(INTEGER VALUE II);
   BEGIN
     IF (PRODUCTION(2,II)=0) AND (PRODUCTION(1,II)<=NT) THEN
       BEGIN INTEGER I,J; I:=1; J:=PRODUCTION(0,II);
       WHILE (I<=6 ) AND (SINGLE(J,I)¬=0) DO I:=I+1;
       IF I=7 THEN WRITE("OOPS1") ELSE SINGLE(J,I):=PRODUCTION(1,II);
       OTHER(J,PRODUCTION(1,II));
       END;
     END;
BEGIN
POINT:=1;
```

```
POINTER:=1;
FOR I:=NT+1 UNTIL NS+1 DO NTERINDEX(I):=0;
FOR I:=1 UNTIL 200 DO NTER(I):=0;
FOR I:=1 UNTIL NS+1 DO
TERINDEX(I):=0;
FOR I:=1 UNTIL  500 DO TER(I):=0;
FOR I:=1 UNTIL NS DO
BEGIN TERINDEX(I):=POINT;
FOR J:=1 UNTIL NP DO BEGIN  INTEGER A,B,C,D,E;
A:=PRODUCTION(2,J);
B:=PRODUCTION(1,J);
C:=PRODUCTION(3,J);
D:=PRODUCTION(4,J);
IF (A=I) OR (B=I) OR (C=I) OR (D=I) THEN
BEGIN TER(POINT):=J; POINT:=POINT+1; END;
END;    END;
FOR I:=NT+1 UNTIL NS DO BEGIN INTEGER B; NTERINDEX(I):=POINTER;
FOR J:=1 UNTIL NP DO BEGIN B:=PRODUCTION(0,J);
IF (B=I) THEN BEGIN NTER(POINTER):=J; POINTER:=POINTER+1; END;
END; END;
NTERINDEX(NS+1):=POINTER;
TERINDEX(NS+1):=POINT;
FOR I:=1 UNTIL NS DO
BEGIN WRITE(" THE FOLLOWING PRODUCTIONS CONTAIN   ",VOCABULARY(I));
FOR I:=TERINDEX(I) UNTIL TERINDEX(I+1)-1 DO WRITE(TER(I));END; END;
FOR I:=NT+1 UNTIL NS DO FOR J:=1 UNTIL 6 DO SINGLE(I,J):=0;
FOR I:=1 UNTIL NP DO SGL(I);
SHIFT;
POINT:=1;
FOR I:=1 UNTIL NS DO BEGIN ONRIGHT(I):=POINT;
FOR K:=TERINDEX(I) UNTIL TERINDEX(I+1)-1 DO BEGIN INTEGER J; J:=TER(K);
                    IF PRODUCTION(4,J)=I THEN BEGIN
RMOST(POINT):=PRODUCTION(0,J); POINT:=POINT+1; END; END;
END;
ONRIGHT(NS+1):=POINT;
FOR I:=NT+1 UNTIL NS DO FOR J:=1 UNTIL 2 DO RIGHTS(I,J):=#0;
FOR I:=NT+1 UNTIL NS DO CURRENT(I):=0;
FOR I:=NT+1 UNTIL NS DO DONE(I):=0;
FOR I:=1 UNTIL NT DO
IF ONRIGHT(I)¬=ONRIGHT(I+1) THEN BEGIN
FOR I:=NT+1 UNTIL NS DO ALRDY(I):=FALSE;
CURSYMBOL:=I; CURSUB:=IF ((I-1) REM 32)=(I-1) THEN 1 ELSE 2; BRS(I);END;
FOR I:=NT+1 UNTIL NS DO FOR K:=1 UNTIL 2*NT DO RIGHTP(I,K):=#0;
   POINT:=1;
   FOR K:=1 UNTIL NS DO
     BEGIN FARRIGHT(K):=POINT;
     FOR I:=TERINDEX(K) UNTIL TERINDEX(K+1)-1 DO
       IF PRODUCTION(4,TER(I))=K THEN
```

```
          BEGIN FRT(POINT):=TER(I); POINT:=POINT+1;
      END;     END;
FARRIGHT(NS+1):=POINT;
FOR I:=1 UNTIL NT DO
      BEGIN CURSYMBOL:=I; CURSUB:=IF (I<=32)              THEN 1 ELSE 2;
      FALNAS(I);
    END;
      READON(FST,SND,TRD);
UNSHIFT;
FOR I:=1 UNTIL NT DO BEGIN CALL:=0; RTSYM:=I;
LOAD(I);
FOR Z:=NT+1 UNTIL NS DO ALRDX(Z):=FALSE; NEED2(I); SORT; END;
END;
              END;
END.
```

LIST OF REFERENCES

Alberga, Cyril N., "String Similarity and Misspellings," Communications of the ACM, v. 10, No. 5, p. 302-313, May 1967.

Blair, Charles R., "A Program For Correcting Spelling Errors," Information and Control, v. 3, No. 1, p. 60-67, March 1960.

Damerau, Fred J., "A Technique for Computer Detection and Correction of Spelling Errors," Communications of the ACM, v. 7, No. 3, p. 171-176, March 1964.

Freeman, D. N., Error Correction in CORC: The Cornell Computing Language, Ph. D. Thesis, Cornell University, Ithaca, New York, 1963.

Gries, David., Compiler Construction for Digital Computers, p. 315-326, Wiley, 1971.

Kildall, Gary, BALGOL-2 User's Manual, Internal Publication Naval Postgraduate School, 1970.

McKeeman, W. M., Horning, J. J. and Wortman, D. B., A Compiler Generator Implemented for the IBM System 360, Prentice Hall, 1970.

Morgan, Howard L., "Spelling Corrections in SYSTEMS PROGRAMS," v. 13, No. 2, p. 90-94, February 1970.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	LT(j.g.) Robert C. Bolles, USNR Code 53Bq Department of Mathematics Naval Postgraduate School Monterey, California 93940	1
4.	ENS Lyle Vernon Rich, USN 1200 North Linden Street Normal, Illinois 61761	1
5.	LCDR Gordon T. McGruther, USN 1022 Spruance Road Monterey, California 93940	1

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author)

Naval Postgraduate School
Monterey, California 93940

2a. REPORT SECURITY CLASSIFICATION

Umclassified

2b. GROUP

3 REPORT TITLE

Error Detection, Analysis and Recovery in XPL Based Compilers

4. DESCRIPTIVE NOTES (Type of report and, inclusive dates)

Master's Thesis; December 1971

5. AUTHOR(S) (First name, middle initial, last name)

Lyle Vernon Rich

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
December 1971	70	8

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Naval Postgraduate School Monterey, California 93940

13. ABSTRACT

This thesis involves the detection, recovery and/or correction of errors in XPL defined languages. XPL is a compiler generating system based on a (1,1) bounded context parser using (2,1) context to resolve conflicts in the grammar, and an analyzer which produces tables from a BNF description of the grammar for the language. The areas of spelling errors and errors caused by insertion/deletion are covered. Routines for correcting spelling errors in an ALGOL-like language are presented. An expanded syntax analyzer which aids in the production of a data base used by the compiler to correct insertion/deletion errors is also presented. Ideas for implementing this data base in XPL compilers, using heuristics to decrease the size of the insertion sets is also presented.

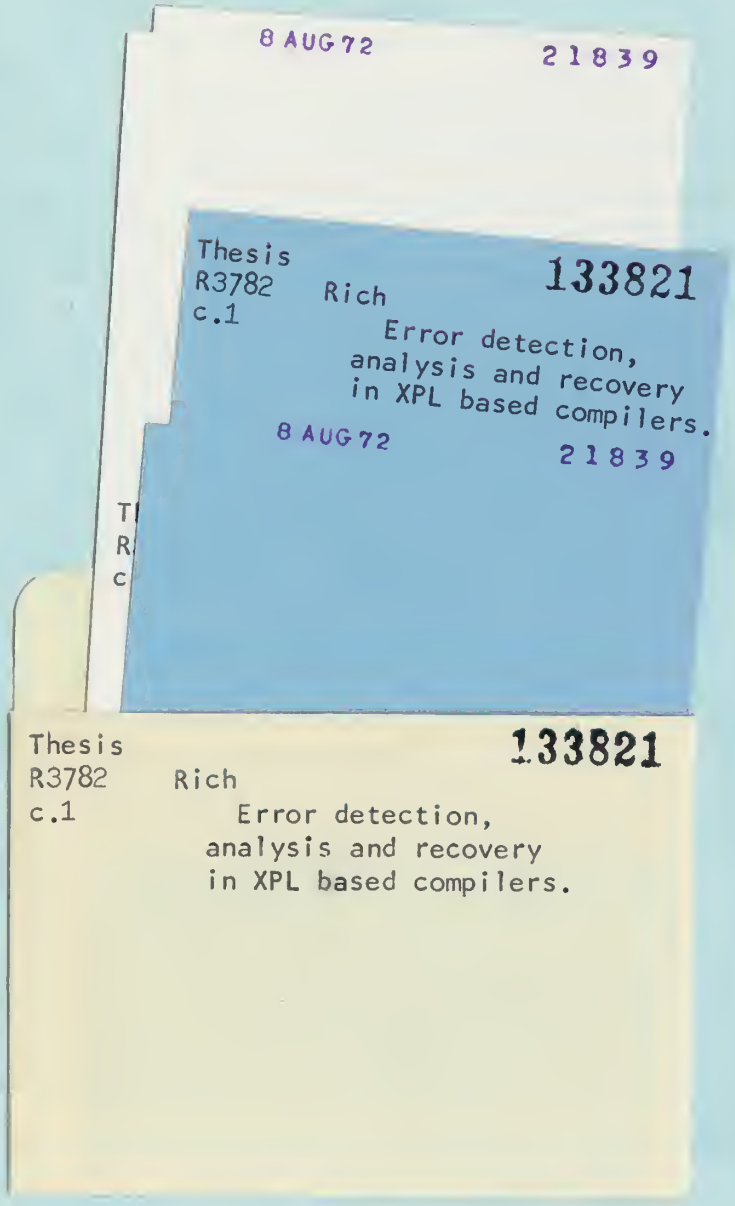
DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6811

Security Classification

A-31408

KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
bounded context parser						
analyzer						
correcting spelling errors						
correct insertion/deletion errors						
heuristics						
XPL						
syntax analyzer						


8 AUG 72 21839
Thesis
R3782 Rich 133821
c.1
Error detection,
analysis and recovery
in XPL based compilers.
8 AUG 72 21839
Thesis
R3782 Rich 133821
c.1
Error detection,
analysis and recovery
in XPL based compilers.

thesR3782
Error detection, analysis and recovery i
3 2768 001 91281 9
DUDLEY KNOX LIBRARY